UNIVERSIDADE FEDERAL DE VIÇOSA

ANÁLISE DE IMPLEMENTAÇÕES DE RANDOM FORESTS EM FPGAS PARA COMPUTAÇÃO EMBARCADA

Alysson Kelvim Caetano da Silva
Magister Scientiae

VIÇOSA - MINAS GERAIS
2024

ALYSSON KELVIM CAETANO DA SILVA

ANÁLISE DE IMPLEMENTAÇÕES DE RANDOM FORESTS EM FPGAS PARA COMPUTAÇÃO EMBARCADA

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

Orientador: Ricardo D Santos Ferreira

Coorientador: Jose Augusto Miranda Nacif

VIÇOSA - MINAS GERAIS
2024

Ficha catalográfica elaborada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa

T
S586a
2024

Silva, Alysson Kelvim Caetano da, 1997-
Análise de implementações de *random forests* em FPGAs para computação embarcada / Alysson Kelvim Caetano da Silva. – Viçosa, MG, 2024.
1 dissertação eletrônica (81 f.): il. (algumas color.).

Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa, Departamento de Informática, 2024.
Referências bibliográficas: f. 76-81.
DOI: https://doi.org/10.47328/ufvbbt.2024.728
Modo de acesso: World Wide Web.

1. Aprendizado do computador. 2. Inteligência artificial. 3. Arquitetura de computador. 4. Arranjos de lógica programável em campo. I. Ferreira, Ricardo dos Santos, 1969-.
II. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22. ed. 006.31

Bibliotecário(a) responsável: Euzébio Luiz Pinto CRB-6/3317

ALYSSON KELVIM CAETANO DA SILVA

ANÁLISE DE IMPLEMENTAÇÕES DE RANDOM FORESTS EM FPGAS PARA COMPUTAÇÃO EMBARCADA

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister*

APROVADA: 9 de julho de 2024.

Assentimento:

_____

Alysson Kelvim Caetano da Silva
Autor

_____

Ricardo Dos Santos Ferreira
Orientador

Essa dissertação foi assinada digitalmente pelo autor em 31/10/2024 às 11:08:22 e pelo orientador em 31/10/2024 às 12:23:10. As assinaturas têm validade legal, conforme o disposto na Medida Provisória 2.200-2/2001 e na Resolução nº 37/2012 do CONARQ. Para conferir a autenticidade, acesse https://siadoc.ufv.br/validar-documento. No campo 'Código de registro', informe o código **NZDB.IJUK.EZTY** e clique no botão 'Validar documento'.

À minha mãe, pelo apoio incondicional, pelas palavras de encorajamento e por acreditar em mim em todos os momentos

AGRADECIMENTOS

Primeiramente, agradeço a Deus, autor da minha existência, por me dar forças e iluminar meu caminho durante toda esta jornada.
A minha mãe, Rose Caetano, que desde cedo me instruiu no caminho da sabedoria e me incentivou a obter conhecimento. Meu esforço e dedicação devo a você.
Ao meu irmão, Álvaro Kalel, pelo suporte inestimável ao longo dessa caminhada.
Ao professor Ricardo, por todo o apoio e orientação durante este trabalho. Sua orientação foi essencial para a realização deste estudo.
Aos colegas de pesquisa, Olavo Barro, Ícaro Moreira e Jerônimo Penha, pela colaboração e pelo compartilhamento de conhecimentos e experiências.
Aos meus amigos Karine Rodrigues, Guilherme Menezes, Matheus Otero, Ana Clara e Cristiane de Jesus pela amizade e pelo suporte constante.
À Universidade Federal de Viçosa, a FAPEMIG e ao EDITAL 001/2022 - DEMANDA UNIVERSAL, APQ-01577-22 PROJETO: “MAPEAMENTO DE ACELERADORES DE ALTO DESEMPENHO EM ARQUITETURAS DE DOMÍNIO ESPECÍFICO”, pela oportunidade e pelos recursos disponibilizados para a realização deste trabalho.
E, por fim, a todos que de alguma forma contribuíram para que este sonho se tornasse realidade. Agradeço imensamente por todo o apoio e incentivo.
O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

O conselho da sabedoria é: procure obter sabedoria; use tudo que você possui para adquirir entendimento. - Provérbio de Salomão

RESUMO

SILVA, Alysson Kelvim Caetano da, M.Sc., Universidade Federal de Viçosa, julho de 2024. **Análise de Implementações de Random Forests em FPGAs para Computação Embarcada**. Orientador: Ricardo Dos Santos Ferreira. Coorientador: Jose Augusto Miranda Nacif.

A inteligência artificial e o aprendizado de máquina, especialmente algoritmos como florestas aleatórias (ou Random Forests), tem ganhado destaque por sua eficácia. A importância de aceleradores de hardware, como GPUs e FPGAs, é crucial para otimizar o desempenho desses algoritmos. A paralelização de processos de inferência melhora significativamente a velocidade e a eficiência, atendendo às demandas do aprendizado de máquina. Este estudo analisa implementações de Random Forest em FPGAs, comparando abordagens com comandos condicionais, multiplexadores, tabelas, equações, técnica BDD, uso de quartis e kmeans para quantização sobre os conjunto de dados de referência na literatura como Susy, Covertype, Adult e Drybean. Visando as implementações de Internet das Coisas (IoT) ou computação de borda, este trabalho avalia dois dispositivos FPGA de baixo custo: Xilinx Artix 7-35T, Sipeed GW2AR-18. Foram analisadas diversas implementações e ferramentas, mostrando que a síntese impacta significativamente a utilização de recursos. Implementações baseadas em tabela, apesar de ser a mais popular na literatura, tem o maior consumo em termos de uso de LUTs, enquanto as implementações Condicional, Multiplexador e Equação são mais eficientes, especialmente as metodologias Condicional e Multiplexador que usam menos LUTs. A vantagem da implementação em Tabela é que os dados das árvores ficam todos em memória, evitando a necessidade de sintetização novamente para mudanças na Random Forest. A quantização por quartis mostrou ser eficiente para redução de LUTs nos FPGAs, utilizando menos recursos do que a quantização por K-means. Este trabalho é um dos primeiros a apresentar uma comparação direta das diversas implementações Random Forest em FPGA, mostrando que existe uma grande variação de custo entre as diversas opções. Também foi realizada a validação experimental com protoboard utilizando o FPGA Tang Nano 20K e ESP8266 demonstrou a viabilidade prática das implementações para popularização dos FPGAs nos ambientes de prototipagem de baixo custo.

Palavras-chave: arquiteturas reconfiguráveis. fpga. random forest. aceleradores em hardware

ABSTRACT

SILVA, Alysson Kelvim Caetano da, M.Sc., Universidade Federal de Viçosa, July, 2024. **Random Forest Implementations on FPGAs for Embedded Computing**. Adviser: Ricardo Dos Santos Ferreira. Co-adviser: Jose Augusto Miranda Nacif.

Artificial intelligence and machine learning, particularly algorithms such as Random Forests, have gained prominence for their effectiveness. The importance of hardware accelerators, such as GPUs and FPGAs, is crucial for optimizing the performance of these algorithms. The parallelization of inference processes significantly improves speed and efficiency, meeting the demands of machine learning. This study analyzes Random Forest implementations in FPGAs, comparing approaches such as conditional commands, multiplexers, tables, equations, BDD techniques, and the use of quartiles and k-means for quantization on reference datasets in the literature, such as Susy, Covertype, Adult, and Drybean. Aiming at Internet of Things (IoT) or edge computing implementations, this work evaluates two low-cost FPGA devices: Xilinx Artix 7-35T and Sipeed GW2AR-18. Various implementations and tools were analyzed, showing that synthesis significantly impacts resource utilization. Table-based implementations, although the most popular in the literature, have the highest consumption in terms of LUT usage, while Conditional, Multiplexer, and Equation implementations are more efficient, especially the Conditional and Multiplexer methodologies that use fewer LUTs. The advantage of the Table implementation is that the tree data is stored entirely in memory, avoiding the need for resynthesis when changes to the Random Forest are required. Quartile quantization proved to be efficient in reducing LUTs in FPGAs, using fewer resources than k-means quantization. This work is one of the firsts to present a direct comparison of various Random Forest implementations in FPGAs, showing a significant cost variation among the different options. Experimental validation with protoboards using the Tang Nano 20K FPGA and ESP8266 demonstrated the practical feasibility of the implementations, promoting the use of FPGAs in low-cost prototyping environments.

Keywords: reconfigurable architecture. fpgas. random forest. hardware accelerators.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

Introdução

A Inteligência Artificial (IA) constitui um campo de estudo dentro da ciência da computação que tem como objetivo a emulação das capacidades cognitivas humanas, englobando processos como aprendizado, raciocínio, evolução e adaptação. Este domínio de pesquisa dedica-se à compreensão dos mecanismos subjacentes ao funcionamento do cérebro humano, com a finalidade de desenvolver programas computacionais que sejam capazes de manifestar comportamentos inteligentes em uma ampla gama de contextos e aplicações (Teixeira, 2019).

A subárea do aprendizado de máquina está intrinsecamente ligada ao campo da IA, representando um dos principais métodos pelo qual os sistemas inteligentes são desenvolvidos atualmente. O aprendizado de máquina tem como objetivo criar métodos capazes de identificar padrões nos dados de forma automática e utilizar esses padrões identificados para prever dados futuros ou outros resultados relevantes. Suas vantagens são diversas e impactam significativamente várias campos da indústria e de pesquisa. Especialmente a capacidade de processar e analisar grandes volumes de dados de forma eficiente e precisa, permitindo a descoberta de percepções que seriam impraticáveis por meio de métodos tradicionais. Além disso, o aprendizado de máquina pode automatizar processos complexos, aumentar a precisão de previsões e recomendações, e adaptar-se a mudanças nos dados de entrada, tornando-se uma ferramenta indispensável para inovação e competitividade em um mercado cada vez mais orientado por dados (Murphy, 2012).

Dentro do universo do aprendizado de máquina, o algoritmo *Random Forest* destaca-se como uma técnica robusta para tarefas de classificação e regressão. *Random Forest* funciona através da construção de múltiplas árvores de decisão durante o treinamento, e da agregação dos resultados dessas árvores para melhorar a precisão das previsões e reduzir o risco de *overfitting*. Este método, ao utilizar um conjunto diversificado de árvores de decisão, combina suas previsões individuais para obter uma decisão final mais confiável e precisa (Murphy, 2012).

O algoritmo *Random Forest* pode ser acelerado explorando a paralelização das árvores usando FPGAs ou GPUs, conforme demonstrado em trabalhos como de Jinguji et al. (2018) e Cheng and Bouganis (2013) e que mostraram uma redução significativa

no tempo de processamento ao utilizar esses aceleradores de hardware em relação às CPUs.

1.1 Motivação

A Figura 1.1 mostra a pesquisa realizada no site de competição de aprendizado de máquina *Kaggle* em 2021, onde vemos que os algoritmos mais comumente utilizados foram a regressão linear e logística, seguidos de perto por árvores de decisão e florestas aleatórias.

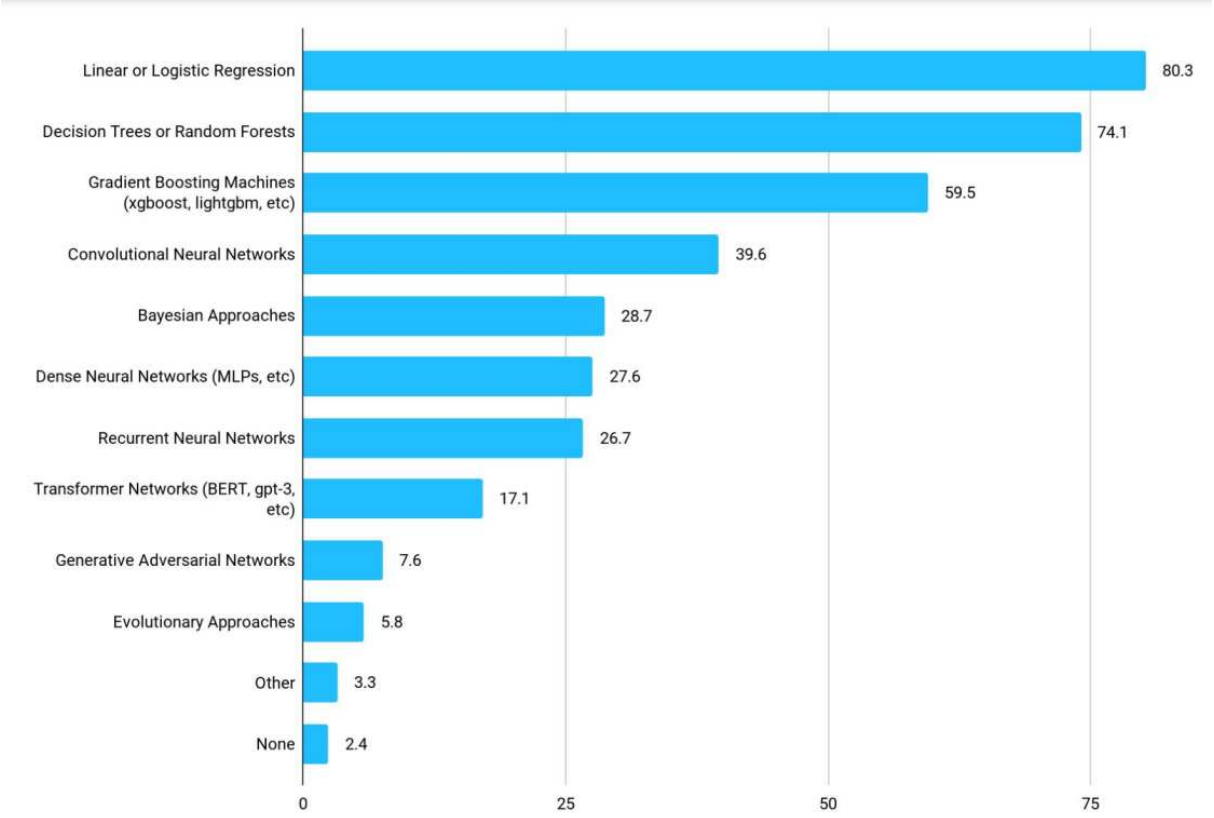


Figura 1.1: Algoritmos mais comumente utilizados em problemas de aprendizado de máquina. Figura extraída e adaptada de Kaggle (2021)

Existe um potencial de paralelismo nos algoritmos de aprendizado de máquina que já foi explorado com sucesso e demonstrou ganhos significativos de desempenho. Por exemplo, a utilização de GPUs para acelerar o treinamento e a inferência de modelos de aprendizado de máquina é uma prática comum que tem mostrado grandes melhorias em termos de velocidade e eficiência. Além disso, técnicas de paralelização têm sido aplicadas em algoritmos como florestas aleatórias para reduzir o tempo de execução e aumentar a escalabilidade.

No entanto, ainda há um grande potencial para melhorias adicionais. No cenário de IoT (Internet das Coisas), onde há uma grande restrição de recursos, podemos

explorar o uso de FPGAs de baixo custo para implementar Random Forest. Um dos desafios é desenvolver implementações eficientes no uso dos recursos dos FPGAs capazes de explorar o paralelismo das Random Forests. FPGAs possuem a capacidade de realizar operações paralelas de maneira eficiente, acelerando algoritmos de aprendizado de máquina em dispositivos com recursos limitados.

Além disso, técnicas de compressão e quantização de modelos podem ser aplicadas para reduzir o uso de memória e a complexidade computacional, tornando-os mais adequados para dispositivos de baixo custo. A exploração dessas tecnologias pode levar a melhorias significativas no desempenho e na eficiência de dispositivos IoT, permitindo a aplicação de modelos complexos de aprendizado de máquina em um ambiente de recursos limitados.

1.2 Justificativa

Apesar de já existirem várias propostas para otimizar algoritmos de aprendizado de máquina, ainda há espaço para melhorias, especialmente quando se trata de adaptar essas otimizações às características específicas dos FPGAs, seja em ambientes de nuvem ou em sistemas embarcados. A necessidade é ainda mais pronunciada em sistemas embarcados que utilizam FPGAs de baixo custo, onde a eficiência no uso de recursos é crucial.

Os FPGAs oferecem uma flexibilidade que traz desafios, especialmente em termos de otimização da quantidade de recursos utilizados, como áreas de lógica, blocos de memória e consumo de energia. A utilização eficiente desses recursos é fundamental para viabilizar o uso de algoritmos complexos em dispositivos de baixo custo, que muitas vezes têm limitações severas em termos de energia e capacidade computacional. Portanto, justifica-se a investigação contínua e aprofundada de métodos para otimizar algoritmos de aprendizado de máquina com foco na utilização eficiente de FPGAs de baixo custo.

1.3 Objetivo Geral

O objetivo geral desta pesquisa é avaliar as diferentes implementações de *Random Forest* propostas para FPGA, compará-las visando otimizar a quantidade de recursos utilizados, principalmente em termos de LUTs (*Look-Up Tables*).

1.3.1 Objetivos Específicos

Os objetivos específicos deste estudo incluem:

- Desenvolver e implementar *Random Forests* utilizando diferentes abordagens, como comandos condicionais, multiplexadores (*Mux*), tabelas e equações, e avaliar qual abordagem apresenta melhor desempenho em termos de tamanho e quantidade de recursos do FPGA utilizados.
- Implementar técnicas de pré-processamento de dados utilizando algoritmos como *K-Means* e *Quartiles*, e avaliar se essas técnicas melhoram o desempenho e a eficiência da *Random Forest*.
- Implementar a técnica BDD (*Binary Decision Diagrams*) para avaliar se esta abordagem resulta em uma utilização mais eficiente dos recursos do FPGA.
- Comparar diferentes implementações em termos de uso de recursos do FPGA, com foco principal na otimização da quantidade de LUTs utilizadas.
- Analisar os resultados das diferentes técnicas e determinar a abordagem mais eficiente para a implementação de *Random Forests* em FPGAs de baixo custo.

Estes objetivos visam contribuir para o desenvolvimento de soluções mais eficientes e viáveis para a implementação de algoritmos de aprendizado de máquina em dispositivos com recursos limitados, promovendo avanços significativos no campo da computação embarcada.

1.4 Problema

Além da proposta de implementação, um projeto em FPGA deve ser especificado usando uma linguagem de descrição de hardware, como Verilog ou VHDL, que será sintetizado e otimizado pela ferramenta de FPGA (Cruz et al., 2022). Os principais fabricantes de FPGA são Intel (Altera) e AMD (Xilinx), mas recentemente surgiram novos fabricantes, como a GoWin, com a ferramenta de síntese GoWin Beta(de Vos et al., 2020). Ferramentas de código aberto, como o pacote Yosys (Wolf et al., 2013), também são uma opção. Um desafio é comparar essas ferramentas para observar como elas interpretam e otimizam a descrição do modelo de implementação.

Explorar diferentes técnicas de implementação para maximizar o desempenho e minimizar o uso de recursos é crucial. Existem várias abordagens para implementar Random Forests, incluindo o uso de comandos condicionais, multiplexadores, tabelas e equações. Determinar qual técnica é mais adequada para FPGAs de baixo custo é um desafio central.

Uma Random Forest pode ser vista como uma função booleana com múltiplas entradas (atributos) e múltiplas saídas (classes) (Bryant, 1986). A ferramenta de síntese transforma essa descrição em um circuito de forma transparente para o usuário. Um

ponto a ser explorado é qual formato será mais otimizado. Recentemente, técnicas usando Binary Decision Diagrams (BDDs) propuseram a transformação das Random Forests em diagramas de decisão binária, que podem ser diretamente mapeados em circuitos (Silva et al., 2023). Nesta dissertação, investigaremos essas abordagens em FPGA e compararemos com abordagens anteriores já avaliadas em FPGA.

A incorporação de técnicas como BDDs, K-Means e quantização de modelos pode melhorar a eficiência de uso dos recursos. No entanto, a eficácia dessas técnicas em FPGAs de baixo custo precisa ser investigada.

1.5 Soluções Propostas

Para enfrentar os desafios identificados, este estudo propõe uma série de soluções estruturadas em torno de técnicas específicas de implementação e otimização para FPGAs de baixo custo. Primeiramente, exploraremos a implementação de *Random Forests* utilizando diferentes abordagens, como comandos condicionais, multiplexadores, tabelas e equações. Cada técnica será avaliada quanto à eficiência no uso de recursos, especialmente LUTs (Look-Up Tables), e ao desempenho geral.

Adicionalmente, técnicas de pré-processamento de dados, como K-Means e quartis, serão implementadas para melhorar a eficiência e a precisão das *Random Forests*. Estas técnicas ajudarão a segmentar e organizar os dados de entrada, potencialmente reduzindo a complexidade do modelo.

Utilizaremos as ferramentas de síntese GoWin da Sipeed e Vivado da Xilinx para simular e analisar o comportamento das implementações em diversos cenários. Em diversos trabalhos anteriores, como o de Wang et al. (2022), foi observado que o uso de conjuntos de dados pequenos com um grande número de árvores não é sempre eficaz. Em cerca de um terço dos casos, a acurácia foi melhor com apenas 10 a 15 árvores, enquanto os demais casos apresentaram ganhos limitados mesmo com um aumento significativo no número de árvores. Outro estudo, de Van Chu et al. (2021) utilizou os datasets Airline, Higgs e Covtype, alcançando acurácia entre 60-70% utilizando de 100 a 200 árvores com profundidade 6 para estes datasets. Portanto, é fundamental determinar um número de árvores que garanta acurácia e seja viável para implementação em FPGAs de baixo custo.

Pra conjuntos de dados pequenos, a utilização de um grande número de árvores é ineficaz e pode causar *overfitting*. A acurácia não melhora significativamente e, em alguns casos, até piora, resultando em um desperdício substancial de recursos. Portanto, é de grande importância, determinar um número de árvores que balanceie eficiência e precisão, especialmente em implementações de baixo custo em FPGAs.

Essas soluções propostas visam abordar de forma abrangente os desafios identificados, buscando maximizar a eficiência e a viabilidade da implementação de al-

goritmos de aprendizado de máquina em FPGAs de baixo custo, contribuindo para avanços significativos no campo da computação embarcada.

1.6 Publicações Associadas ao Desenvolvimento da Pesquisa

Ao longo do desenvolvimento desta dissertação, alguns trabalhos foram publicados, tanto como autor principal quanto como coautor, que estão diretamente alinhados com os objetivos e contribuições deste estudo. A seguir, são apresentados e contextualizados esses trabalhos, evidenciando sua relação com a temática central desta pesquisa.

O trabalho "Implementações eficientes de random forest em FPGA de baixo custo para internet das coisas e computação de borda"(Silva et al., 2024), investiga abordagens eficientes de implementação do algoritmo Random Forest em FPGAs de baixo custo. Este estudo é relevante para a dissertação, pois explora técnicas que otimizam recursos computacionais, essenciais para aplicações de borda e IoT, áreas centrais da pesquisa. O artigo compara implementações utilizando diferentes descrições em FPGAs e destaca a importância de escolher a abordagem de acordo com a ferramenta de síntese empregada. Além disso, analisa o impacto de técnicas como quantização e algoritmos híbridos, o que contribui para os estudos relacionados à otimização de modelos de aprendizado de máquina.

O artigo "RDSF: Everything at Same Place All at Once – A Random Decision Single Forest"(Silva et al., 2023), propõe um novo modelo gráfico para representar Random Forests, chamado Random Decision Single Forest (RDSF). Esta contribuição é especialmente relevante para a dissertação, pois o RDSF utiliza diagramas de decisão binária (BDD) para lidar com desafios típicos das implementações tradicionais de Random Forest. O método oferece melhorias significativas em termos de escalabilidade e tempo de execução, aspectos cruciais para otimizar algoritmos em contextos de alto desempenho, como os discutidos nesta pesquisa.

O estudo "Avaliação de Estilos de Código para Árvores de Decisão em GPU com Microbenchmarks"(Silva et al., 2022), investiga o desempenho de árvores de decisão implementadas em GPUs. A relevância deste estudo para a dissertação reside na análise detalhada de como diferentes estilos de implementação impactam o desempenho em diferentes níveis de profundidade. Os resultados mostram que implementações sem desvio são mais eficientes para árvores rasas, enquanto para profundidades maiores, o uso de desvios é mais vantajoso, apesar das divergências. A compreensão dessas nuances de performance é fundamental para esta pesquisa, que visa otimizar algoritmos em plataformas heterogêneas.

Por fim, o artigo "Hpyc-FPGA: Integração de Aceleradores em FPGA de Alto Desempenho com Python para Jupyter Notebooks"(Silva et al., 2022), investiga a integração de aceleradores em FPGA utilizando o ambiente PYNQ da Xilinx e Jupyter Notebooks. Este trabalho se relaciona com a dissertação ao abordar a aplicação de FPGAs para aceleração de algoritmos, utilizando como estudo de caso o algoritmo K-means. A pesquisa demonstra que a integração de FPGA com Python pode proporcionar ganhos de desempenho significativos em comparação com sistemas tradicionais baseados em processadores Xeon, destacando a importância de explorar plataformas de hardware alternativas para processamento eficiente.

Essas publicações, ao longo do desenvolvimento da dissertação, contribuem diretamente para a compreensão e o avanço das técnicas de otimização em aprendizado de máquina e plataformas heterogêneas, oferecendo bases sólidas para as propostas e resultados apresentados nesta pesquisa.

1.7 Estrutura da Dissertação

O restante desta dissertação está organizada em outros quatro capítulos, conforme descrito a seguir:

O segundo capítulo aborda os fundamentos teóricos necessários para compreender o desenvolvimento deste trabalho. São discutidos os conceitos de Inteligência Artificial (IA) e aprendizado de máquina, incluindo uma introdução às árvores de decisão e ao algoritmo de *Random Forest*. Adicionalmente, são explicadas as técnicas de *K-Means* e quartis, que desempenham um papel importante no pré-processamento de dados. O capítulo também inclui uma discussão sobre hardware, detalhando o funcionamento de GPUs e FPGAs, bem como suas linguagens de programação e ferramentas de síntese.

No terceiro capítulo, são descritas as principais implementações de *Random Forest* para FPGAs, analisando diferentes abordagens como comandos condicionais, multiplexadores, tabelas e equações. Este capítulo também explora a técnica de *Binary Decision Diagrams* (BDD) e apresenta uma revisão da literatura relacionada, discutindo como estudos anteriores implementaram essas técnicas e os resultados obtidos.

O quarto capítulo apresenta os resultados da pesquisa, detalhando os *datasets* e dispositivos utilizados nas experimentações. São discutidos os recursos de FPGA consumidos por cada *dataset* e as diferentes implementações analisadas. Este capítulo também inclui uma comparação entre diversas ferramentas de síntese, destacando a eficiência de cada abordagem.

Finalmente, o quinto capítulo conclui a dissertação, resumindo os principais achados da pesquisa e suas implicações. São discutidas as contribuições do estudo para o campo da computação embarcada e sugeridas direções para trabalhos futuros.

Capítulo 2

Fundamentos

2.1 Aprendizado de Máquina

O aprendizado de máquina constitui um campo da inteligência artificial que se concentra no desenvolvimento de algoritmos e modelos capazes de aprender e tomar decisões com base em dados observados, sem que sejam explicitamente programados para tarefas específicas. Este campo busca identificar padrões em conjuntos de dados e construir modelos matemáticos a partir desses padrões, os quais são utilizados para realizar previsões ou tomar decisões sobre novos dados ainda não observados.

Existem diversas abordagens e técnicas dentro da aprendizagem de máquina, sendo as principais a aprendizagem supervisionada e a aprendizagem não supervisionada (Prince, 2023). Nesta dissertação, exploraremos as técnicas de aprendizado supervisionado, especificamente o Random Forest, e de aprendizado não supervisionado, como o K-means. A técnica K-means será empregada para a redução de dimensionalidade. Uma vez que essa redução seja realizada, o modelo resultante poderá ser modelado utilizando o aprendizado supervisionado por meio do Random Forest, como iremos demonstrar, gerando um exemplo de Random Forest com dois níveis.

2.1.1 Aprendizado Supervisionado

O aprendizado supervisionado refere-se a métodos onde os modelos são treinados em conjuntos de dados rotulados, o que significa que cada exemplo de treinamento possui uma correspondência conhecida entre os dados de entrada e as saídas desejadas. Durante o treinamento, os algoritmos ajustam seus parâmetros para minimizar o erro nas previsões com base nos rótulos fornecidos (Dhankhad et al., 2018).

Algoritmos de aprendizagem supervisionada incluem a regressão linear, que visa encontrar a linha que melhor se ajusta a um conjunto de pares de valores. Além disso, redes neurais, que são modelos inspirados no funcionamento do cérebro humano, e *Random Forest*, que é um conjunto de árvores de decisão, também são exemplos de algoritmos de aprendizado supervisionado. Aplicações práticas do aprendizado su-

pervisionado abrangem áreas como reconhecimento de padrões, detecção de fraudes financeiras, análise de sentimentos em textos e tradução automática (Balahur and Turchi, 2014; Courtney et al., 2020).

2.1.2 Aprendizado Não Supervisionado

A aprendizagem não supervisionada envolve a extração de padrões ou estruturas ocultas a partir de conjuntos de dados não rotulados, isto é, conjuntos de dados para os quais não há informações prévias sobre as categorias ou classes dos dados. Um dos principais métodos de aprendizado não supervisionado é a clusterização, que tem como objetivo agrupar os dados em clusters ou grupos baseados em similaridades intrínsecas entre os pontos de dados. O algoritmo *K-means* é um exemplo clássico utilizado para esse fim (Chong et al., 2021).

Após os trabalhos de Chong et al. (2021), diversas implementações de *K-means* utilizando aceleradores em GPU ou FPGA foram desenvolvidas Penha et al. (2018); Bragança et al. (2021); da Silva Alves et al. (2023); Bueno et al. (2024a), com o objetivo de melhorar o desempenho em comparação às implementações tradicionais baseadas em CPU.

2.2 Árvore de Decisão

Árvore de decisão é uma das técnicas de aprendizado de máquina mais conhecidos e simples. Conhecidas por sua estrutura gráfica intuitiva e capacidade de destacar atributos relevantes para a classificação. Essas árvores, em geral, simplificam a análise ao incorporar apenas um subconjunto de atributos, em vez de todos, permitindo que os usuários concentrem-se nos mais significativos. A estrutura hierárquica das árvores de decisão fornece informações sobre a importância relativa dos diferentes atributos (Freitas, 2014). Por exemplo, podemos criar uma árvore de decisão para a classificação de insetos; um exemplo de pesquisa, ao mesmo tempo didático, está disponível em Marulanda Lopez et al. (2024) e pode ser manipulado usando Google Colab para ensino e pesquisa Canesche et al. (2021).

Os atributos considerados nas árvores de decisão podem ser numéricos ou categóricos. Os atributos numéricos são tratados com base em valores de corte, que são determinados de maneira a maximizar a separação entre as diferentes classes. Estes valores de cortes são escolhidos de modo a dividir o conjunto de dados de forma que cada subconjunto seja o mais homogêneo possível em relação à variável de saída. Por outro lado, os atributos categóricos são tratados agrupando-se os dados com base em suas categorias distintas, facilitando a identificação de padrões específicos a cada categoria. (Leo Breiman, 1984)

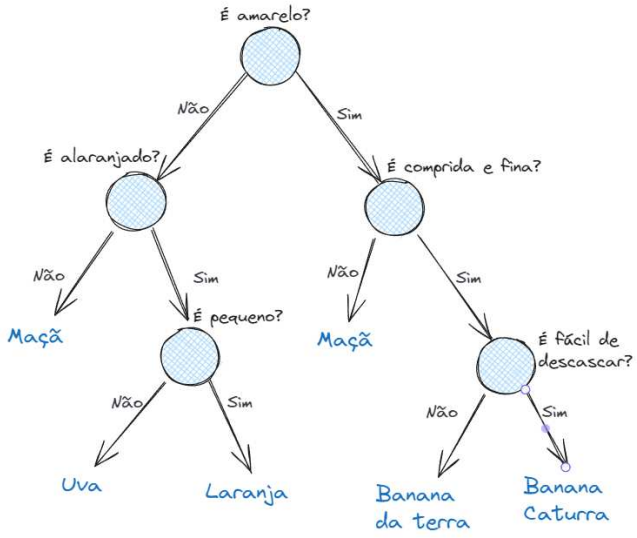


Figura 2.1: Figura extraída e adaptada de Hartshorn (2016)

A Figura 2.1 exemplifica a aplicação prática de uma árvore de decisão no contexto de classificação de frutas, uma tarefa comum em aprendizado de máquina. Na representação visual, são destacados atributos chave como cor e formato, que são essenciais para distinguir entre frutas como laranja, maçã, uva, banana da terra e banana caturra. Cada nó da árvore corresponde a uma pergunta decisiva — como 'é amarelo?', 'é alaranjado?', 'é comprida e fina?', 'é pequeno?' ou 'é fácil de descascar?' — que guia o processo de classificação. O exemplo ilustra claramente a metodologia de dividir o conjunto de dados em subconjuntos mais homogêneos, com base nas respostas a estas perguntas, facilitando a identificação precisa e rápida da fruta em análise. Tal estrutura não apenas torna o processo de decisão transparente, mas também revela a importância relativa de cada atributo na determinação do resultado final, alinhando-se perfeitamente com os princípios fundamentais das árvores de decisão em aprendizado de máquina.

A árvore é construída a partir de uma tabela, neste exemplo, os atributos podem ser a cor, o formato, o tamanho, a casca, entre outros. O exemplo ilustrou apenas dados categóricos. As folhas representam as classes temos 5 classes. Outro ponto é a profundidade da árvore que neste exemplo é no máximo igual a três.

2.2.1 Construção de uma Árvore de Decisão

A construção de uma árvore de decisão é caracterizada pela divisão recursiva de um conjunto de dados em subconjuntos mais homogêneos. Este processo inicia no nó raiz e se estende até os nós folha. A ideia central é criar divisões que maximizem a pureza de cada subconjunto, resultando em uma estrutura que pode ser facilmente interpretada e aplicada em tarefas de classificação e regressão. (Loh, 2011)

Essas divisões são baseadas em critérios que são usados para avaliar e selecionar as melhores divisões em cada nó ao construir árvores de decisão para tarefas de classificação e regressão.

O índice de Gini, mais comumente utilizado em árvores de decisão, é uma métrica usada para avaliar a impureza ou heterogeneidade de um conjunto de dados, sendo comumente utilizado na construção de árvores de decisão. Sua fórmula é expressa como $Gini = 1 - \sum_{i=1}^{n} p_i^2$, onde p_i representa a proporção de amostras da classe i dentro do nó. O valor do índice de Gini varia entre 0 e 1: um valor de 0 indica pureza total (ou seja, todas as amostras pertencem à mesma classe), enquanto um valor de 1 representa impureza máxima.

O cálculo do índice de Gini pode ser ilustrado utilizando o exemplo de cinco tipos de frutas: laranja, maçã, uva, banana da terra e banana caturra. Para um primeiro exemplo, considere a seguinte distribuição: (0, 10, 20, 3, 5), com 38 amostras.

Primeiramente, calcula-se a proporção de amostra para cada classe e então, calcula-se o índice de Gini:

$$
\begin{aligned}
p_1 &= \frac{0}{38} = 0 \\
p_2 &= \frac{10}{38} \approx 0,263 \\
p_3 &= \frac{20}{38} \approx 0,526 \\
p_4 &= \frac{3}{38} \approx 0,079 \\
p_5 &= \frac{5}{38} \approx 0,132
\end{aligned}
$$

Índice de Gini:

$$
\begin{aligned}
Gini &= 1 - (0^2 + 0,263^2 + 0,526^2 + 0,079^2 + 0,132^2) \\
Gini &\approx 1 - 0,369 = 0,631
\end{aligned}
$$

Dessa forma, obtém-se o índice de impureza de 0,631. Para observar o comportamento do índice de Gini com amostras mais puras, considere a seguinte distribuição: (0, 50, 1, 1, 1), com 53 amostras. Neste exemplo, a segunda classe possui notavelmente mais amostras do que as demais:

$$p_1 = \frac{0}{53} = 0$$
$$p_2 = \frac{50}{53} \approx 0,943$$
$$p_3 = \frac{1}{53} \approx 0,019$$
$$p_4 = \frac{1}{53} \approx 0,019$$
$$p_5 = \frac{1}{53} \approx 0,019$$

Índice de Gini:

$$Gini = 1 - (0^2 + 0,943^2 + 0,019^2 + 0,019^2 + 0,019^2)$$
$$Gini \approx 1 - 0,89 = 0,11$$

Neste caso, obtém-se um índice de impureza significativamente menor do que no exemplo anterior, com valor de 0,11. Para analisar um último exemplo, considere uma distribuição onde as classes estão bem distribuídas: (10, 10, 10, 10, 10), totalizando 50 amostras.

$$p_1 = \frac{10}{50} = 0,2$$
$$p_2 = \frac{10}{50} = 0,2$$
$$p_3 = \frac{10}{50} = 0,2$$
$$p_4 = \frac{10}{50} = 0,2$$
$$p_5 = \frac{10}{50} = 0,2$$

Índice de Gini:

$$Gini = 1 - (0,2^2 + 0,2^2 + 0,2^2 + 0,2^2 + 0,2^2)$$
$$Gini = 1 - 0,2 = 0,8$$

Esses exemplos demonstram como diferentes distribuições de classes dentro de um nó afetam a impureza do nó e, consequentemente, a decisão de como particionar os dados ao construir uma árvore de decisão. Uma menor impureza (índice de Gini mais baixo) indica uma maior assertividade no nó, que é o objetivo ao se criar subdivisões durante a construção da árvore. O índice de Gini é particularmente útil tanto

para variáveis categóricas quanto numéricas, tornando-o uma ferramenta versátil para particionar dados em algoritmos de árvore de decisão. (Leo Breiman, 1984)

A árvore é construída a partir da tabela. Primeiro calcula-se o gini para todos os atributos (ou colunas da tabela). O nó raiz será o que tiver melhor Gini e depois vamos descendo recursivamente para construir os outros níveis. A construção pode ser limitada pela profundidade, quantidade mínima de amostras ou quando encontrar uma separação com pureza. Em geral se usa um método guloso na construção. Mas o problema de encontrar a melhor árvore é NP-Completo (Laurent and Rivest, 1976).

2.2.2 Profundidade da Árvore

A profundidade de uma árvore refere-se ao número de níveis ou divisões na estrutura da árvore. Na construção de uma árvore de decisão, a árvore é particionada recursivamente em subconjuntos menores com base nos valores das características, com cada divisão criando um novo nível na árvore. A profundidade da árvore é determinada pelo número de níveis ou divisões necessários para separar os dados em subconjuntos homogêneos com base na variável alvo. Uma árvore rasa possui menos níveis e é menos complexa, enquanto uma árvore profunda possui mais níveis e é mais complexa. (Leo Breiman, 1984)

Controlar a profundidade das árvores é crucial na construção de árvores de decisão para prevenir o *overfitting* e melhorar a generalização do modelo para dados não vistos. É fundamental gerenciar a profundidade das árvores para encontrar um equilíbrio entre capturar padrões significativos nos dados de treinamento e evitar a incorporação de ruídos ou anomalias. Árvores excessivamente profundas podem levar ao *overfitting*, onde o modelo aprende a classificar ou prever com base em peculiaridades específicas dos dados de treinamento ao invés de padrões reais. Ao controlar a profundidade das árvores, pode-se garantir que o modelo capture relações e características essenciais da base de dados sem ser excessivamente sensível a flutuações menores nos dados de treinamento, melhorando assim sua capacidade de fazer previsões precisas em novas instâncias não vistas. Em geral, problemas com muitas classes requerem uma profundidade maior para separar as classes de forma adequada, garantindo a discriminação eficaz entre elas (Leo Breiman, 1984).

Podemos observar que as árvores podem ser desbalanceadas, este fato deve ser considerado pelas implementações em hardware, como veremos mais adiante.

2.2.3 Classificação e Regressão

Árvores de decisão são modelos versáteis que podem ser usados tanto para tarefas de classificação quanto de regressão. No contexto da classificação, árvores de decisão

são empregadas para particionar o espaço de características em regiões que correspondem a diferentes classes ou categorias. Ao dividir recursivamente o espaço de características com base nos valores das características de entrada, as árvores de decisão criam uma estrutura hierárquica de nós de decisão e nós folha, onde cada nó folha representa um rótulo de classe. Isso permite que as árvores de decisão façam previsões sobre a pertença de classe de novas instâncias, seguindo o caminho do nó raiz até um nó folha específico com base nos valores das características do ponto de dados de entrada. (Leo Breiman, 1984)

Em tarefas de regressão, árvores de decisão são utilizadas para prever variáveis-alvo contínuas, particionando o espaço de características e atribuindo um valor numérico a cada nó folha. Semelhante à classificação, as árvores de decisão dividem recursivamente o espaço de características com base nas características de entrada, mas na regressão, os nós folha contêm valores numéricos previstos em vez de rótulos de classe. Ao fazer previsões para novas instâncias, as árvores de decisão as atribuem a nós folha específicos com base em seus valores de características e fornecem os valores numéricos previstos correspondentes associados a esses nós folha. Portanto, árvores de decisão servem como ferramentas poderosas tanto para classificação quanto para regressão, fornecendo uma estrutura flexível e interpretável para modelagem preditiva em vários domínios. (Leo Breiman, 1984)

Nesta dissertação, abordaremos problemas de classificação utilizando aceleradores baseados em FPGA. No entanto, é importante notar que modelos de regressão também podem se beneficiar significativamente dessas tecnologias. Estudos como o de Gunasekaran et al. (2021) utilizaram a técnica de *random forest* em um modelo de regressão para selecionar atributos de conjuntos de dados para serem utilizados um sistema de monitoramento de tráfego de rede em tempo real baseado em FPGA. Além deste, Pham-Quoc et al. (2024) apresentaram uma arquitetura que acelera o algoritmo *random forest* em plataformas de computação de borda na execução de modelos preditivos de regressão.

2.3 Random Forest

Random Forests são um tipo de algoritmo de aprendizado de máquina que utiliza múltiplas árvores de decisão para realizar a classificação ou regressão de dados. Cada árvore de decisão é construída a partir de uma amostra aleatória dos dados de treinamento e, em seguida, as previsões de cada árvore são combinadas para produzir uma previsão final mais precisa. As *Random Forests* são frequentemente utilizadas em problemas de classificação, como reconhecimento de imagens, detecção de fraudes e diagnóstico médico. Elas são conhecidas por sua capacidade de lidar com dados complexos e de alta dimensionalidade, além de serem resistentes a *overfitting*. (Hartshorn,

2016)

Durante o treinamento, antes de construir cada árvore de decisão na floresta, uma amostra aleatória dos dados de treinamento é selecionada. Com esta amostra, uma árvore de decisão é construída utilizando um subconjunto dos atributos de modo que cada árvore é treinada para realizar previsões com base nos atributos disponíveis. Durante o treinamento, é comum aplicar técnicas para controlar o *overfitting*, ajustando os chamados hiperparâmetros que são utilzados para limitar a profundidade das árvores, estabelecer um número mínimo de amostras para dividir um nó e utilizar diferentes amostras aleatórias em cada árvore além de definir uma quantidade de árvores a ser utilizado no treinamento. Esses passos garantem que as *Random Forests* sejam capazes de capturar a complexidade dos dados de treinamento e produzir previsões robustas e precisas para novos dados. (Hartshorn, 2016)

O treinamento e a inferência das *random forests* podem ser realizados em paralelo. Dado um novo conjunto de dados, cada árvore na floresta é utilizada para fazer uma previsão com base nos atributos disponíveis. Em seguida, as previsões de cada árvore são combinadas para produzir uma previsão final. Na tarefa de classificação, a previsão final pode ser determinada por votação majoritária, ou seja, a classe mais frequente entre as previsões das árvores é escolhida como a previsão final. A Figura 2.2 ilustra a votação em uma *random forest* para a classificação de frutas, onde cada uma das quatro árvores da floresta faz uma previsão baseada nos atributos disponíveis: a primeira árvore vota em uva, enquanto a segunda e a terceira votam em laranja, e a última em banana da terra. Neste caso, a laranja é declarada a vencedora por ser a mais votada, ou seja, a classe mais frequente entre as previsões das árvores é adotada como a previsão final. Na tarefa de regressão, a previsão final pode ser a média das previsões das árvores. É importante notar que, durante a inferência, cada árvore na floresta é utilizada de forma independente, o que permite que a *random forest* seja facilmente paralelizada e escalonada para grandes conjuntos de dados. (Hartshorn, 2016)

2.4 Algoritmo K-means

O algoritmo *K-means* é uma técnica de clusterização amplamente utilizada para agrupar conjuntos de dados não rotulados em *K clusters* distintos. Este algoritmo é particularmente valorizado pela sua simplicidade e eficiência na segmentação de dados.

O procedimento do K-means inicia-se com a seleção aleatória de *K* centroides, que representam os centros iniciais dos *clusters*. Cada ponto de dados é então atribuído ao *cluster* cujo centróide está mais próximo, com base em uma métrica de distância, geralmente a distância euclidiana ou, em alguns casos específicos, a distância de Hamming. Após essa atribuição inicial, os centroides de cada *cluster* são recalculados

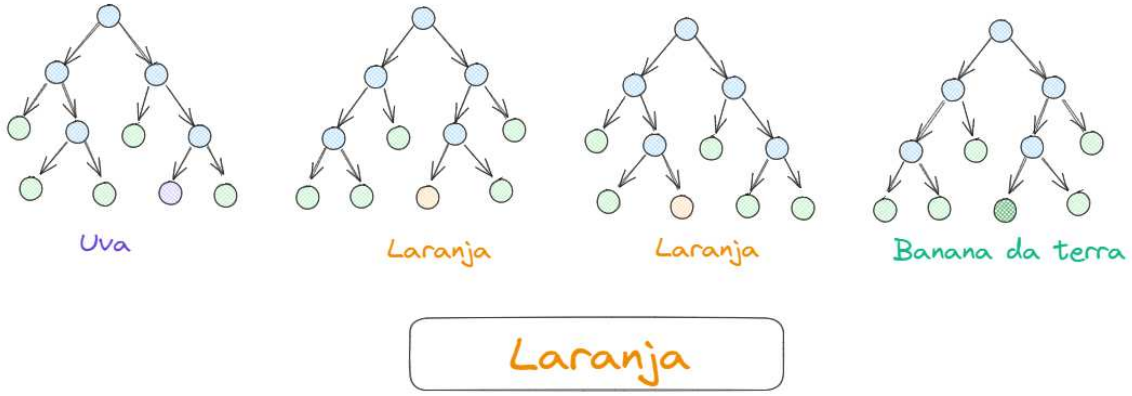


Figura 2.2: Votação Majoritária em *Random Forest* para Classificação de Frutas.

como a média dos pontos atribuídos a esse *cluster*. Este processo de atribuição e atualização dos centroides é repetido iterativamente até que a convergência seja alcançada, ou seja, até que não haja mais mudanças significativas na atribuição dos pontos aos *clusters* ou até que um número máximo de iterações seja atingido (Chong et al., 2021).

Em muitas dimensões é difícil visualizar o funcionamento de algoritmo. Vamos considerar um problema com apenas dois atributos (ou dimensões) como ilustrado na Figura 2.3, o algoritmo K-means distribui os dados em *clusters* distintos, sendo visualmente evidente a separação dos grupos, neste caso para $k = 3$, ou seja, 3 grupos.

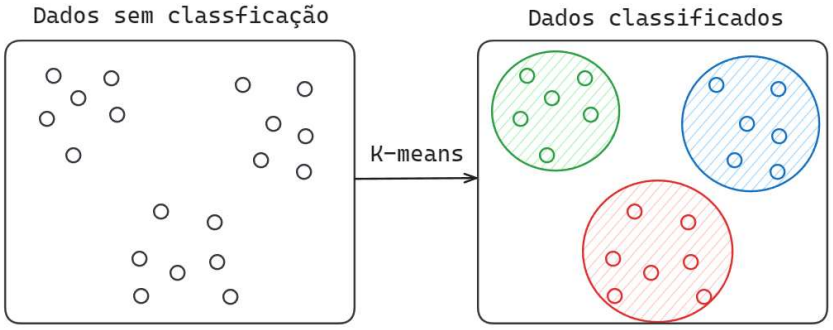


Figura 2.3: Ilustração de um agrupamento de dados utilizando K-means com três clusters.

O objetivo fundamental do algoritmo K-means é minimizar a variância *intra-cluster*, definida como a soma das distâncias ao quadrado entre cada ponto de dados e seu centróide atribuído. Essa minimização é crucial para garantir que os pontos dentro de um mesmo *cluster* sejam o mais semelhantes possível entre si, e, ao mesmo tempo, que os diferentes *clusters* sejam tão distintos quanto possível.

Apesar de sua eficácia e fácil implementação, o algoritmo K-means possui algumas limitações. Primeiramente, é sensível à inicialização dos centroides, o que pode levar a diferentes resultados em execuções distintas do algoritmo. Além disso, a determina-

ção do número ideal de *clusters K* é um desafio que geralmente requer conhecimento prévio do domínio dos dados ou a utilização de métodos heurísticos, como o método do cotovelo (*elbow method*).

Além de sua ampla aplicação em tarefas de agrupamento, o algoritmo K-means é frequentemente utilizado para quantização de conjunto de dados. A quantização é o processo de mapear um conjunto grande e contínuo de valores para um conjunto menor e discreto, o que é particularmente útil em compressão de dados e em simplificação de modelos de dados.

Por exemplo, considere um conjunto de dados com valores contínuos que precisam ser reduzidos para um número limitado de representações distintas. O K-means pode ser usado para identificar os agrupamentos e seus centróides, que servirão como os valores a serem quantizados. Cada ponto de dado no conjunto de dados é então substituído pelo rótulo do seu grupo ou *cluster*, reduzindo assim a dimensão dos dados enquanto busca preservar a informação essencial. No exemplo da Figura 2.3 podemos codificar os dois atributos com 2 bits para os três grupos: 00 para verde, 01 para azul e 10 para o vermelho, por exemplo.

Esse método pode ser ilustrado com um exemplo prático. Suponha que temos um conjunto de dados com valores contínuos de temperaturas medidas ao longo de um dia, como mostrado na tabela abaixo:

Temperatura Original (°C)
21.5
23.0
22.8
20.1
19.6
24.5
25.0
23.5

Tabela 2.1: Valores originais de temperaturas

Aplicando o algoritmo K-means para quantizar esses valores em 3 *clusters*, obtemos os seguintes centroides: 20.1, 22.8, e 24.5. A tabela abaixo mostra como os valores originais das temperaturas são substituídos pelos seus respectivos centroides que podem ser codificados com 2 bits:

Neste exemplo, os valores contínuos das temperaturas são mapeados para um conjunto discreto de valores representados pelos centroides, facilitando a compressão de dados.

Temperatura Original (°C)	Temperatura Quantizada (°C)
21.5	22.8 ou código 01
23.0	22.8 ou código 01
22.8	22.8 ou código 01
20.1	20.1 ou código 00
19.6	20.1 ou código 00
24.5	24.5 ou código 10
25.0	24.5 ou código 10
23.5	24.5 ou código 10

Tabela 2.2: Valores quantizados de temperaturas usando $K = 3$, ou seja 3 grupos que podem ser codificados com 2 bits

2.5 Quartis

Os quartis são medidas estatísticas fundamentais utilizadas para dividir um conjunto de dados em quatro partes iguais, cada uma representando 25% dos dados. Esses pontos de divisão são chamados de primeiro quartil (Q1), segundo quartil (Q2) e terceiro quartil (Q3), e cada um deles oferece uma visão específica sobre a distribuição dos dados.

O primeiro quartil (Q1) é o valor abaixo do qual se encontra a primeira parte de quatro dos dados ordenados. Em outras palavras, 25% dos dados são menores ou iguais a este valor. O segundo quartil (Q2), também conhecido como mediana, representa o ponto central da distribuição dos dados. Isto significa que 50% dos dados são menores ou iguais a este valor, dividindo assim a distribuição em duas metades iguais. O terceiro quartil (Q3) é o valor abaixo do qual está o terceiro quarto dos dados ordenados, indicando que 75% dos dados são menores ou iguais a este valor (Zaki and Meira, 2014). A figura 2.4 ilustra como esta distribuição é dada.

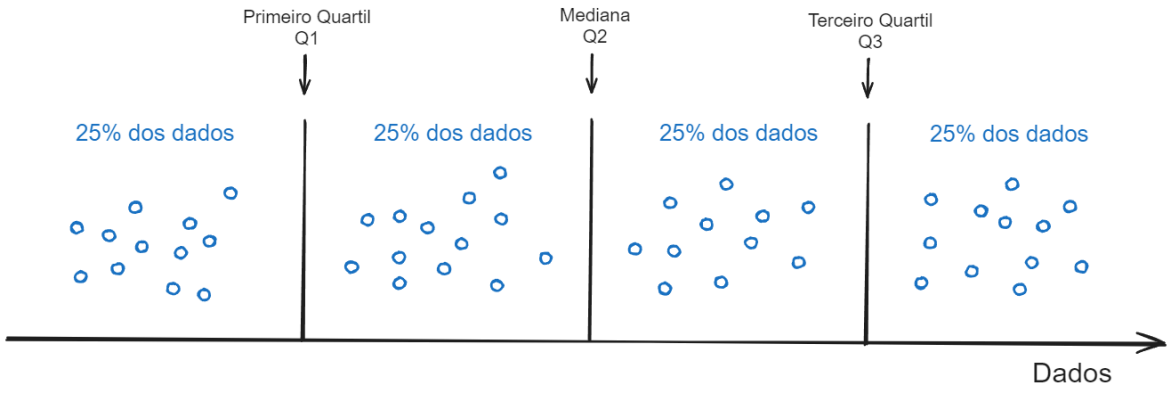


Figura 2.4: Distribuição dos dados utilizando quartis.

Os quartis fornecem uma maneira robusta e intuitiva de resumir e interpretar a

distribuição de dados, sendo ferramentas indispensáveis em estatísticas descritivas e análise exploratória de dados. Usando quartis podemos, por exemplo, codificar os dados com 2 bits, um código para cada quartil.

2.6 Unidades de Processamento Gráfico (GPUs)

As Unidades de Processamento Gráfico (GPUs) são aceleradores de *hardware* desenvolvidos inicialmente para aplicações que envolvem processamento gráfico intensivo. Projetadas como uma arquitetura altamente paralela, as GPUs são compostas por milhares de núcleos de processamento, conhecidos como *CUDA cores* na NVIDIA ou *Stream Processors* na AMD. Esses núcleos são organizados em blocos de processamento denominados *Streaming Multiprocessors* (SMs) na arquitetura da NVIDIA, cada um capaz de executar múltiplas operações simultaneamente. Essa capacidade de processamento paralelo permite que as GPUs lidem com grandes volumes de dados de forma eficiente, proporcionando um desempenho significativamente superior em comparação com as Unidades Centrais de Processamento (CPUs) tradicionais.

Uma das principais características das GPUs é a capacidade de realizar cálculos de ponto flutuante de alta precisão, o que é essencial para operações matemáticas complexas envolvidas em gráficos 3D, simulações físicas e aprendizado de máquina. As GPUs modernas suportam operações de ponto flutuante de precisão simples (32 bits) e dupla (64 bits), permitindo cálculos precisos e eficientes em uma ampla gama de aplicações .

Outro aspecto técnico importante das GPUs é a capacidade de execução assíncrona e o gerenciamento eficiente de *threads*. Isso permite que as GPUs processem múltiplas tarefas simultaneamente, aproveitando o paralelismo em diferentes partes do processo de renderização ou processamento computacional. Esse gerenciamento eficiente de *threads* é fundamental para maximizar o desempenho em tarefas de alta intensidade computacional (Cheng et al., 2014).

Entretanto, a implementação de *random forest* pode não ser tão eficiente pois a tradução natural de uma árvore gera um conjunto de IF aninhados. Os comandos IF geram divergência de controle que impacta no desempenho das GPUs (Cheng et al., 2014). Em um estudo recente do grupo de pesquisa da Universidade Federal de Viçosa (Penha et al., 2023), mostramos que até a profundidade de 7 níveis, as implementações em GPU pode ser eficientes sem uso de IF, com mais de 7 níveis a melhor opção são os IF, mesmo com divergências. Além disso, uma implementação com as árvore codificadas como tabelas não é interessante, devido a grande latência da memória da GPU, mesmo usando as memórias dedicadas de constante ou memória compartilhada.

Alguns trabalhos destacam o uso das GPUs para construção das *random forest*

na etapa de treinamento, aproveitando seu potencial de paralelismo para otimizar o desempenho. Esta dissertação foca na etapa de inferência. Para treinamento, as GPUs oferecem bastante potencial como por exemplo, os algoritmos gpuRF e gpuERT foram desenvolvidos para construção de árvores de decisão em GPUs (Jansson et al., 2014). Da mesma forma, a implementação paralela para construção Random Forest na etapa de treinamento em GPU mostrou um aumento significativo de desempenho e aceleração, especialmente em conjuntos de dados de baixa dimensionalidade (Senagi and Jouandeau, 2022).

2.7 Computação Reconfigurável

Os processadores de propósito geral são projetados para executar uma ampla variedade de tarefas computacionais, sendo amplamente utilizados em sistemas embarcados, *desktops*, servidores e dispositivos móveis. No entanto, para aplicações específicas que requerem desempenho otimizado e flexibilidade, os processadores de propósito geral podem não ser a melhor escolha. Neste contexto, a computação reconfigurável surge como uma alternativa, oferecendo a capacidade de adaptar a arquitetura de hardware às necessidades específicas de uma aplicação (Ordoñez et al., 2005).

2.7.1 Dispositivos FPGAs

Field-programmable gate arrays (FPGAs), ou arranjo de portas programáveis em campo, são chips que, ao contrário dos ASICs (*application-specific integrated circuits*), são reconfiguráveis, permitindo a implementação de qualquer lógica de um circuito digital. Um FPGA é composto por diversos blocos lógicos, que incluem *flip-flops*, LUTs (*look-up tables*) e multiplexadores. Esses blocos lógicos são interconectados através de *switches* em uma grade, possibilitando a criação de circuitos digitais complexos (Boutros and Betz, 2021). Os FPGAs evoluíram significativamente ao longo dos últimos trinta anos. As gerações mais recentes incluem blocos com maior granularidade do que as LUTs, como os DSPs (unidades de cálculo geralmente de 16 a 18 bits) e os blocos dedicados de memória RAM ou BRAM. Como o escopo dessa dissertação são os FPGAs de baixo custo, nesta seção iremos destacar apenas as características básicas dos FPGAs.

Blocos Lógicos

Os blocos lógicos são formados por LUTs, *flip-flops* e multiplexadores, conforme ilustrado na Figura 2.5 sendo responsáveis por implementar uma vasta gama de funções lógicas combinacionais. As LUTs conseguem implementar qualquer função booleana correspondente ao número de suas entradas. No projeto de um elemento lógico (LE),

o tamanho da LUT resulta em um *trade-off* entre velocidade e custo de área. Quanto maior o número de entradas de uma LUT, maior a complexidade das funções booleanas que ela pode representar, reduzindo assim a complexidade em termos de níveis lógicos no caminho crítico do circuito. Contudo, o custo em termos de área de silício consumida aumenta de forma não linear (Ordoñez, 2003). Inicialmente, as LUTs tinham de 3 a 4 entradas. Atualmente os FPGAs modernos possuem LUTs de 6 a 8 entradas.

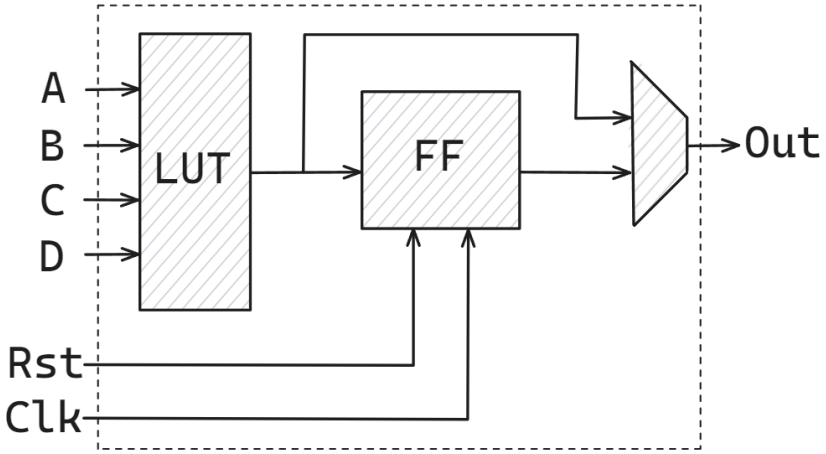


Figura 2.5: Bloco lógico composto por uma LUT de 4 entradas, um *flip-flop* e um multiplexador

Os registradores dentro dos blocos lógicos desempenham a função de elementos de memória, consistindo em uma abstração de um ou mais *flip-flops*. Eles são essenciais para manter a sincronização e assegurar que as operações ocorram de forma ordenada e previsível, representando o estado do circuito (Boutros and Betz, 2021).

Uma LUT também pode ser usada como uma memória. Muitas ferramentas de síntese para FPGA oferecem recursos, seja por meio de bibliotecas na descrição do *hardware* ou por opções de otimização, que permitem utilizar as LUTs como pequenos blocos de memória com 1 *bit* de largura.

Roteamento

Para que um circuito digital possa interconectar todos os blocos lógicos utilizados, é necessário um sistema de roteamento eficiente. O roteamento em um dispositivo FPGA consiste em blocos de *switches* e segmentos de fios com conexões dinâmicas, permitindo a conexão entre os blocos lógicos(Ordoñez, 2003). Ou seja, em um FPGA, tanto a lógica é reconfigurável (LUTs) como as conexões (roteamento).

Os FPGAs utilizam *switches* programáveis para realizar essa tarefa, controlando o fluxo de *bits* para os blocos lógicos corretos. A Figura 2.6 ilustra possibilidades de roteamento com a programação dos *switches*. Um dos maiores desafios hoje para

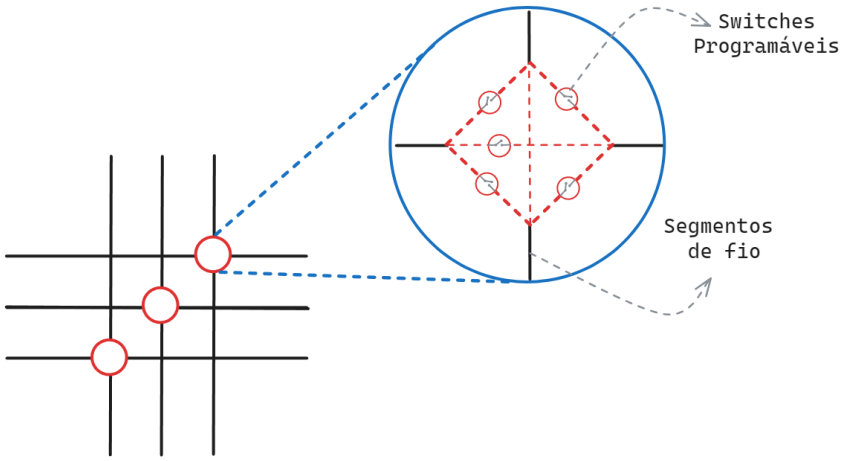


Figura 2.6: Bloco de *switch* para roteamento no FPGA

os FPGAs de alto desempenho são as tarefas de posicionamento e roteamento que podem consumir horas. Neste trabalho usamos FPGAs de baixo custo e complexidade, onde o posicionamento e roteamento pode ser realizado em poucos segundos ou alguns minutos no máximo.

2.7.2 Linguagem de Descrição de Hardware

Assim como a complexidade da programação de *software* levou a uma evolução e variedade de linguagens de programação, na área de descrição de *hardware* tornou-se necessário desenvolver abordagens de programação que possibilitassem a especificação de *hardware* de diversas formas. A linguagem de descrição de *hardware* é transformada de arquivo textual em um grafo (ou *netlist*), ou mapeada em um grafo, que representa o circuito para uma determinada plataforma. Esta etapa é a síntese que realiza o mapeamento tecnológico da descrição de *hardware*. As ferramentas de síntese mais conhecidas em FPGA são Xilinx Vivado e Intel Quartus. Neste trabalho, pelo primeira vez, a ferramenta Gowin é avaliada de forma acadêmica e comparada com as ferramentas comerciais bem conhecidas. As linguagens de descrição de *hardware* costumam suportar as três abstrações mais comuns: RTL (*register transfer level*), comportamental e nível de portas.

Utilizando o nível de abstração RTL, controla-se a transferência de dados entre registradores. O RTL descreve como os dados são movidos de um registrador para outro em resposta a eventos de *clock*, permitindo descrever a lógica que carrega um valor em um registrador quando uma borda de *clock* é detectada. Isto é essencial para projetar e simular circuitos digitais síncronos de forma eficiente.

Na modelagem comportamental, o circuito é descrito a partir de operações de alto nível, como adição, subtração e multiplicação, sem a necessidade de especificar diretamente a implementação de *hardware* em níveis mais detalhados, como registradores

ou portas lógicas. O *hardware* é descrito com base em seu funcionamento, abstraindo detalhes de baixo nível.

Na modelagem a nível de porta (*gate-level*), o *hardware* é descrito através das conexões de portas lógicas, exprimindo o comportamento de um circuito digital em um nível muito próximo da implementação física do *hardware*. Este nível de abstração permite um detalhamento extremamente alto do projeto, possibilitando ajustes com máxima precisão, embora a complexidade do design aumente consideravelmente (Rushton, 2011).

VHDL

O VHDL foi originalmente desenvolvido como parte do programa VHSIC (*Very High Speed Integrated Circuit*), lançado pelo Departamento de Defesa dos Estados Unidos nos anos 1980, com o objetivo de padronizar e melhorar a eficiência do projeto de circuitos digitais complexos. Foi padronizado pela IEEE como IEEE Standard 1076 em 1987 (Rushton, 2011).

Verilog

A linguagem Verilog foi criada em 1983 por Prabhu Goel e Phil Moorby para a Automated Integrated Design Systems, adquirida posteriormente pela Cadence Design Systems. Em 1995, Verilog foi padronizada pela primeira vez como IEEE Standard 1364. Sua sintaxe, baseada na linguagem C, torna a linguagem mais acessível para os usuários. Além disso, devido à sua padronização, há uma ampla variedade de compiladores e simuladores disponíveis. A linguagem Verilog foi projetada com foco na modelagem a nível de porta, apresentando uma sintaxe intuitiva para construtos de baixo nível (Rushton, 2011; Smith, 1996). Neste trabalho optamos por construir geradores com a linguagem Verilog, que é mais simples, mais próxima da linguagem C e vem se popularizando nos últimos anos e substituindo o uso de VHDL na academia e na industria.

High-Level Synthesis (HLS)

High-Level Synthesis (HLS) refere-se ao uso de linguagens de programação de alto nível, como C ou C++, com a adição de pragmas para definir a síntese de hardware. O HLS permite a descrição de algoritmos em um nível mais abstrato, facilitando a transição do *software* para o *hardware*. Essa abordagem tem sido utilizada em diversos trabalhos para a implementação de algoritmos de aprendizado de máquina, como *random forest*, devido à sua capacidade de gerar *hardware* eficiente e otimizado. Trabalhos recentes têm explorado o uso de HLS para melhorar o desempenho e a eficiência de sistemas de aprendizado de máquina em hardware (Liu et al., 2019; Arapidis, 2024).

É importante notar que, neste trabalho, o foco será na investigação do nível *Register Transfer Level* (RTL) utilizando Verilog, devido à sua maior compacidade e controle sobre os detalhes de implementação do hardware.

2.8 Aceleradores em Hardware

Aceleradores em hardware, também conhecidos como co-processadores, são chips que executam uma ou mais tarefas específicas, aliviando a CPU dessas funções. Isso permite que a CPU aloque seu tempo de forma mais eficiente para tarefas mais diversificadas que não necessitam de um *hardware* dedicado. Além disso, devido à menor complexidade de um *hardware* específico em comparação com uma CPU, a vazão de dados é consideravelmente maior e o consumo energético é reduzido.

Devido à equivalência *hardware-software*, qualquer algoritmo baseado em CPU pode ser implementado como um circuito digital. Por exemplo, algoritmos de criptografia podem ser executados por meio da abordagem de criptografia baseada em *hardware* (*hardware-based encryption*), que auxilia ou substitui totalmente o *software* no processo de criptografia. Os algoritmos de criptografia possuem complexidade computacional não polinomial, tornando-se vantajoso o uso de aceleradores de *hardware* para aumentar a eficiência e a segurança (Dipu Kabir and Alam, 2010).

O processamento gráfico é uma das tarefas mais comuns para aceleradores de *hardware*, com a GPU sendo o tipo mais difundido. As GPUs são essenciais para aplicações que demandam alto desempenho gráfico, como jogos, renderização de imagens e aplicações de aprendizado de máquina (Cheng et al., 2014).

Aceleradores também são amplamente utilizados para o processamento de pacotes de rede devido à alta demanda por processamento de pacotes TCP/IP em servidores. A utilização de um *hardware* específico para essa tarefa é atraente, pois pode melhorar significativamente o desempenho e a eficiência energética dos sistemas de rede (Ruiz et al., 2019).

A Figura 2.7 ilustra a utilização de um acelerador como co-processador, destacando a interação entre a CPU e o acelerador para a execução de tarefas específicas.

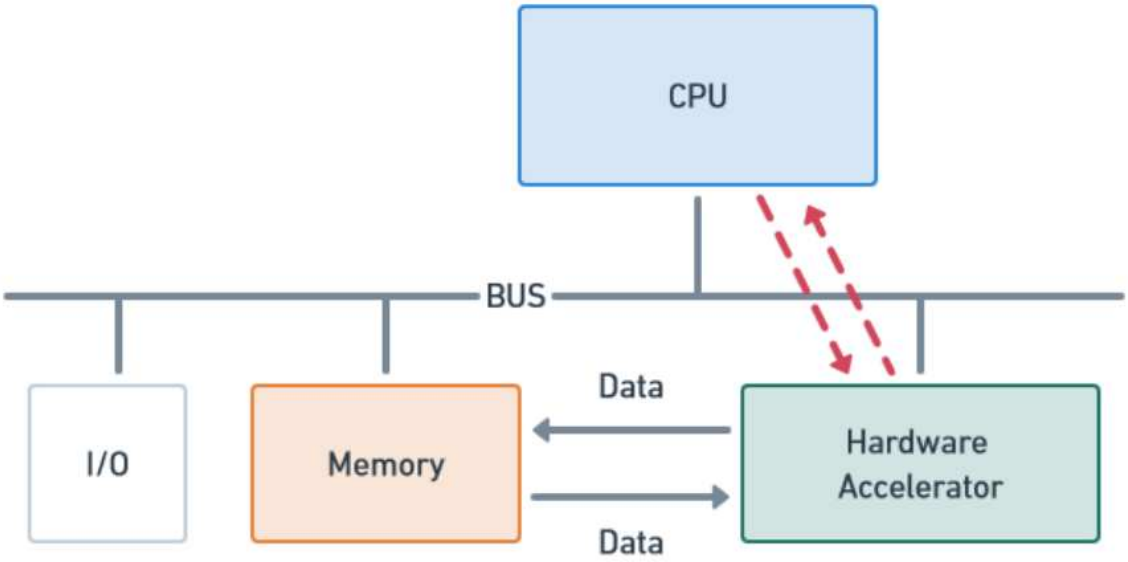


Figura 2.7: Acelerador como co-processador com memória separada

Capítulo 3

Implementações

Este capítulo apresenta implementações de inferências nas *Random Forest*, explorando diferentes abordagens para otimizar a execução em hardware. Serão abordados métodos utilizando comandos condicionais, tabelas de inferência, árvores com multiplexadores, equações booleanas, e Diagramas de Decisão Binárias (BDDs). Além disso, discutiremos a soma de bits para votação e os trabalhos relacionados.

3.1 Comandos Condicionais e Multiplexadores

Em Verilog, é possível utilizar duas abordagens distintas para a descrição de um multiplexador: através da estrutura condicional comportamental "if-else"ou por meio do operador condicional ternário. Inicialmente, será apresentada a descrição utilizando a estrutura "if-else"e, em seguida, será demonstrado o uso do operador ternário para a mesma finalidade.

Uma implementação direta de uma árvore de decisão pode ser descrita com uma sequência de comandos "if-else"aninhados. Os operadores condicionais, devido à sua natureza binária, são adequados para este propósito, aceitando ou rejeitando a expressão contida. Em cada nó da árvore, uma condição é avaliada no comando "if". Se a condição for verdadeira, os blocos aninhados dentro do corpo do "if"são avaliados e executados sequencialmente. Caso contrário, se a condição resultar em falso, o bloco "else"é executado. Este processo continua de forma aninhada, do mais externo ao mais interno, até encontrar um nó folha, que representa a classe final. No caso de uma *random forest*, cada árvore individual incrementa um contador de classe. Após a execução de todas as árvores, o passo seguinte é determinar qual contador tem o maior valor, o que definirá a classificação final gerada.

Na Figura 3.3, apresenta-se uma árvore de decisão balanceada representada na forma de um aninhamento de estruturas condicionais. A raiz começa computando o primeiro comparador. Neste exemplo, há duas variáveis, X1 e X2, ambas representando entradas. As variáveis são comparadas de forma que, caso obedeçam à condição que torna a expressão verdadeira, o nó à direita, representando outro bloco condicional, é executado.

Na implementação utilizando o operador condicional "?", que explicitamente descreve um multiplexador, cada nó de decisão é representado por um multiplexador. Para simplificar o circuito responsável por determinar a classe vencedora, a saída de cada árvore pode ser codificada em formato *one-hot*.

Neste método, a condição ou seletor do multiplexador é comparador para um atributo específico atende a um determinado limiar (*threshold*). A implementação pode usar dois níveis. Primeiro usamos uma camada de comparadores e depois que é repassada para um segundo nível onde conectamos com os seletores dos multiplexadores.

A figura 3.1 ilustra a relação entre uma árvore de decisão e seus circuitos comparadores correspondentes. Do lado esquerdo, temos uma árvore de decisão com três nós de comparação, cada um utilizando um atributo diferente: a1, a2 e a3. Ao lado direito, encontram-se três circuitos comparadores, cada um vinculado a um nó da árvore e colorido da mesma forma para indicar essa associação. Nos circuitos comparadores, os valores dos atributos a1, a2 ou a3 são inseridos, ocorrendo uma comparação que resulta em uma saída binária (verdadeiro ou falso, representado por 0 ou 1) resultando nas saídas s1, s2 e s3, respectivamente, para cada comparador. Esses resultados, s1, s2 e s3, serão utilizados posteriormente como seletores nos multiplexadores.

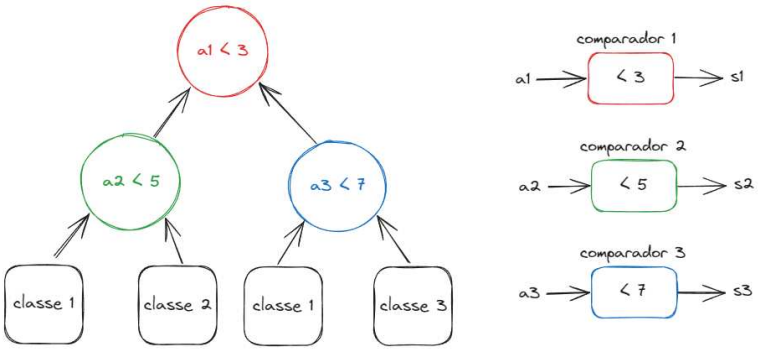
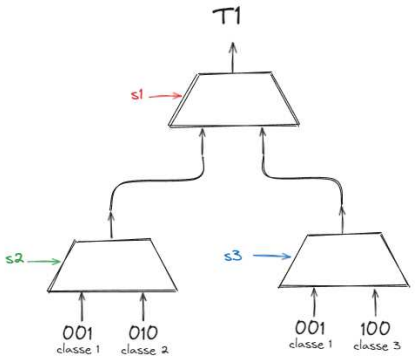


Figura 3.1: Árvore de Decisão e Circuitos Comparadores Correspondentes.

Com os circuitos comparadores devidamente configurados, procede-se à construção da estrutura da árvore de decisão, empregando multiplexadores. Esta estrutura opera de maneira ascendente, isto é, de baixo para cima. Nas entradas dos multiplexadores, são introduzidas as diversas classes, as quais são subsequentemente encaminhadas conforme as avaliações realizadas pelos comparadores. À medida que a entrada progride através dos multiplexadores, ocorre a filtragem e direcionamento das classes com base nos resultados obtidos pelos comparadores. Esta sequência de operações culmina na classificação final de uma dada amostra, que é efetuada pela árvore em questão.



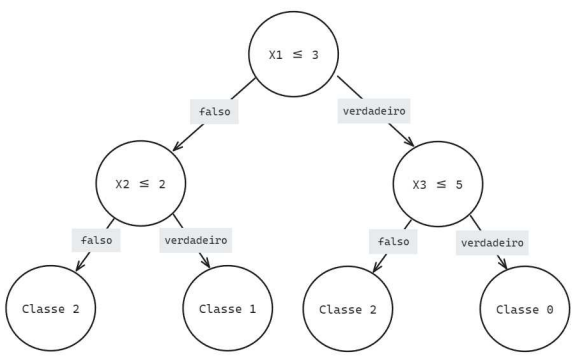

```
//Pseudocódigo para 3 classes
reg [2:0] T1; //One hot
reg s1,s2,s3;

//Comparadores
s1 = (a1>3);
s2 = (a2>5);
s3 = (a3>7);

//Árvore
T1 = s1 ?
    (s2 ? classe1 : classe2) :
    (s3 ? classe1 : classe3);
```

Figura 3.2: Representação de Multiplexadores e Pseudocódigo para Classificação da Árvore de Decisão.

A figura 3.2 ilustra a estrutura de multiplexadores e um pseudocódigo correspondente, que juntos representam o funcionamento de uma árvore de decisão para classificação em três classes. Do lado esquerdo, estão três multiplexadores utilizando os seletores s1, s2 e s3, derivados dos circuitos comparadores descritos anteriormente. Cada multiplexador recebe como entrada as classes 1, 2 e 3, definidas pela árvore de decisão, e produz um resultado denominado T1, que simboliza a saída da primeira árvore.

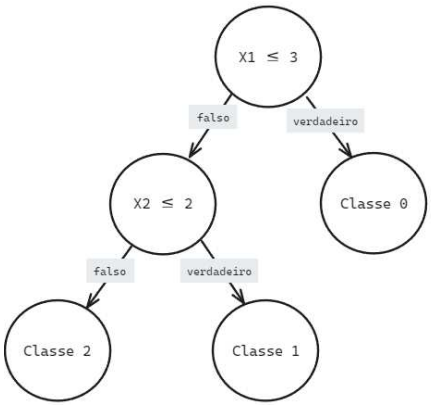
A descrição utilizando o operador condicional "?"é mais explícita para a ferramenta de síntese. O código exemplificado não apresenta nenhum controle síncrono. No entanto, tanto os comparadores quanto as árvores podem incorporar um controle em dois níveis: no comparador e na saída da árvore. A estrutura comportamental "if-else"também pode implementar uma camada de comparadores e uma execução em pipeline, consistindo de uma etapa de comparação seguida pela inferência na árvore.

No lado direito, o pseudocódigo detalha o processo de classificação para três classes. Ele inicia declarando T1 como um registrador de 3 bits (indicado por [2:0]), com a representação em *one-hot* das classes. A codificação *one-hot* é um método de representação de estados onde apenas um *bit* está ativo (ou 'alto') por vez, tornando a identificação de classes exclusiva e clara. Por exemplo, a classe 1 pode ser representada como '001', a classe 2 como '010' e a classe 3 como '100'. Os seletores s1, s2 e s3 são definidos com base nas comparações de atributos a1, a2 e a3 com seus respectivos limites (*thresholds*).




```
input wire [2:0] x1;
input wire [2:0] x2;
input wire [2:0] v;
always @(posedge clock)
begin
    if (x1 ≤ 3) begin
        if(x3 ≤ 5) begin
            v ⟸ class0;
        end
        else begin
            v ⟸ class2;
        end    end
    else begin
        if(x2 ≤ 2) begin
            v ⟸ class1;
        end
        else begin
            v ⟸ class2;
        end
    end
end
```

Figura 3.3: Representação de uma árvore de decisão balanceada utilizando blocos "if-else".




```
input wire [2:0] x1;
input wire [2:0] x2;
input wire [2:0] v;
always @(posedge clock) begin
    if (x1 ≤ 3) begin
        v ⟸ class0;
    end
    else begin
        if(x2 ≤ 2) begin
            v ⟸ class1;
        end
        else begin
            v ⟸ class2;
        end
    end
end
```

Figura 3.4: Representação de uma árvore de decisão desbalanceada utilizando blocos "if-else".

No fragmento de código Verilog representado nas Figuras 3.3 e 3.4, ilustra-se um exemplo do processo de inferência. O registrador v representa a classe votada, com sua largura definida para se adequar à representação *one-hot*, em que somente um *bit* do registrador será 1 (verdadeiro) por vez, sendo a posição deste *bit* representativa da classe inferida. A codificação *one-hot* é conveniente pois elimina a necessidade de um teste de igualdade entre o valor da saída e um valor de comparação para aferir o resultado; basta verificar o estado de cada *bit* individualmente para determinar a classe inferida.

3.2 Tabelas de Inferência

Uma árvore pode ser descrita em formato de tabela, com cada nó representado em uma linha. Essa tabela pode ser armazenada em memória, o que facilita a reconfiguração dinâmica do circuito. Nas implementações condicionais, seja comportamental ou estrutural, uma vez sintetizadas, não é possível modificar a implementação. A tabela

permite o uso de um único comparador (versão serial) ou de vários comparadores (no caso de *pipeline*). No caso de pipeline, o circuito poderá resultar em um roteamento mais complexo dos dados de entrada, conforme veremos a seguir.

A natureza reconfigurável da implementação baseada em tabela implica um custo benefício, aumentando a complexidade do circuito. Isso ocorre porque o projeto do caminho de dados deve ser independente tanto dos comparadores quanto dos dados.

3.2.1 Árvore de Decisão

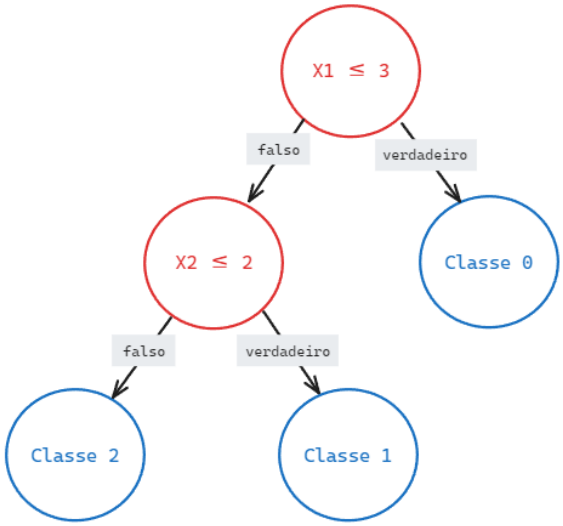
A Figura 3.5 apresenta uma representação de uma pequena tabela de inferência, que corresponde a uma árvore com cinco nós. Para representar uma árvore de decisão em formato de tabela, é essencial considerar dois tipos distintos de nós: os nós intermediários, que contêm comparações, e os nós folhas, que contêm as classes. Em função disso, o tipo de nó altera a semântica de cada coluna da tabela. Por essa razão, na Figura 3.5 são utilizados dois cabeçalhos: o cabeçalho em vermelho, correspondente às colunas para um nó interno, e o cabeçalho em azul, correspondente às colunas para um nó folha, ou seja, podemos definir que um campo da tabela tenha mais de uma significado em função do tipo do nó: interno (vermelho) ou folha (em azul).

Para determinar se um nó é interno ou folha, a primeira coluna reserva um *bit* para servir como *flag*. Quando se trata de nós internos, é necessário armazenar o limiar a ser comparado, ou seja, o valor a ser utilizado na comparação. Por exemplo, se a comparação for $x_1 \leq 3$, o limiar é 3. A coluna seguinte armazena o índice da coluna do conjunto de dados que será utilizado na comparação. Se o conjunto de dados possui três colunas, os índices podem ser x_0, x_1 ou x_2. Esse índice indica qual dado da amostra deve ser comparado em determinado nó. Após as comparações, é preciso definir o destino, seja verdadeiro ou falso; assim, as duas colunas subsequentes armazenam o índice de memória que indica o próximo nó caso a comparação seja falsa ou verdadeira.

No caso dos nós folhas, além da *flag* de indicação do tipo de nó, é necessário apenas armazenar a classe correspondente a esse nó e o índice da próxima árvore, para reiniciar as comparações até a conclusão.

O processo de inferência pode realizado sequencialmente ou através de um pipeline. As árvores podem ser avaliadas sequencialmente ou em paralelo, ao final do processo a classe mais votada pela floresta é calculada. Ao alcançar um nó folha, um registrador é incrementado, correspondendo à classe votada. Ao final do processo os registradores são comparados para verificar qual possui o maior valor, definindo assim a classe vencedora da floresta.

O uso de tabelas também pode ser implementado em uma versão pipeline. Nesse caso, é mais eficiente ter uma tabela por nível, uma estratégia frequentemente utili-



	(Interno ou Folha) (Interno ou Folha)	Limiar –	Coluna Comparada –	Índice próx. nó se falso Índice prox. árvore	Índice próx. nó se verdadeiro Classe
0	Interno	3	1	1	2
1	Interno	2	2	3	4
2	Folha	–	–	5	0
3	Folha	–	–	5	2
4	Folha	–	–	5	1
5	Próxima Árvore				

Figura 3.5: Representação de uma tabela de inferência mostrando uma árvore de decisão com cinco nós.

zada em implementações em FPGA. Saqib et al. (2013) propuseram uma arquitetura onde os dados são distribuídos em um pipeline para execução simultânea, o que reduz os ciclos de *clock* necessários para a classificação, especialmente em grandes conjuntos de dados. O trabalho de Struharik (2011) examina diferentes arquiteturas, incluindo aquelas baseadas em pipelines. Nesse estudo, cada nível da árvore de decisão é associado a um estágio de pipeline, com cada estágio correspondendo a um nível da árvore de decisão. A desvantagem dessa abordagem é a necessidade de rotear todos os dados, pois não se sabe qual caminho será seguido. No entanto, uma vantagem é que não é necessário realizar todas as comparações.

Entretanto, existem trabalhos que preferem usar uma camada de comparadores e na tabela armazenar apenas o índice do comparador. Como Owaida et al. (2017), em que as unidades de processamento das árvores de decisão consistem em quatro operações: ler um nó da árvore da memória da árvore; ler a característica correspondente do exemplo de dado da memória de dados; comparar o limiar do nó da árvore com o valor da característica; e, em seguida, calcular o ponteiro do próximo nó de decisão ou ler o nó folha. Essas são as operações necessárias para avaliar um nível da árvore

para o exemplo de dado de entrada. Para avaliar todos os níveis da árvore, o ponteiro do próximo nó de decisão é realimentado para a primeira operação para continuar o processamento dos níveis subsequentes. Essas quatro operações são iteradas até que um nó folha seja alcançado.

```
proximoIndice = 0;

para cada amostra {
    noAtual = tabela[proximoIndice];

    se noAtual.tipoDoNo == "interno" {
        se x[noAtual.colunaComparada] ≤ noAtual.limiar {
            proximoIndice = noAtual.indiceSeVerdadeiro;
        } senão {
            proximoIndice = noAtual.indiceSeFalso;
        }
    } senão {
        // nó folha
        classe[noAtual.classe]++;
        proximoIndice = noAtual.proximaArvore;
    }
}
```

Figura 3.6: Pseudocódigo do processamento da tabela.

Na Figura 3.6, o pseudocódigo implementa a navegação em uma árvore de decisão utilizando uma estrutura de armazenamento baseada em tabelas. Em uma árvore de decisão, cada nó interno representa uma condição de decisão baseada em um limiar aplicado a uma característica (coluna) específica dos dados de entrada. Quando um nó interno é encontrado, o código verifica se o valor da característica atual da amostra é menor ou igual ao limiar. Dependendo do resultado, o próximo índice é atualizado para seguir o caminho da condição verdadeira ou falsa. Se um nó folha é encontrado, a contagem da classe correspondente é incrementada e o próximo índice é atualizado para continuar a avaliação em outra árvore, se houver.

Como já mencionado, uma das grandes vantagens de utilizar tabelas para armazenar a estrutura das árvores de decisão é a flexibilidade que isso proporciona. Quando a árvore de decisão é representada em uma tabela de memória, não é necessário reconfigurar o FPGA cada vez que se deseja utilizar uma árvore diferente. Em vez disso, pode-se simplesmente atualizar o conteúdo da memória com os novos valores que representam a nova árvore.

3.3 Equações Booleanas

Uma árvore de decisão pode ser representada como uma função booleana, onde as entradas ou variáveis primárias são os comparadores e as saídas representam a codificação das classes. Considerando a codificação *one-hot*, cada bit corresponde a uma

função booleana de uma classe específica. Para a construção das equações de cada classe, conforme detalharemos, é necessário percorrer todos os caminhos que conduzem até a classe em questão.

A abordagem de equações booleanas envolve operadores lógicos básicos, como E (*AND*), OU (*OR*) e negações (*NOT*). Neste método, toda a árvore é abstraída em n equações booleanas, sendo n o número de classes, com cada equação representando o caminho que leva à inferência dessa classe. Diferentemente de outros métodos onde cada nó é representado por uma abstração individual, todas as comparações dos limiares com as entradas são calculadas antecipadamente e representam as variáveis. Por fim, os resultados das n equações são salvos em um registrador de n *bits* que utiliza a codificação *one-hot* para posteriormente calcular o resultado da inferência de toda a floresta (Wang et al., 2022).

A Figura 3.7 mostra uma árvore de decisões destacando os caminhos relativos para cada uma das classes (0, 1 e 2) usando equações booleanas. Na Figura 3.7(a), podemos observar que existem dois caminhos possíveis para inferir a classe 0. No primeiro caminho, a primeira entrada ($e1$) é menor que 1, resultando em um valor lógico verdadeiro, e a segunda entrada ($e2$) não é menor ou igual a 4. Assim, o *bit* de saída correspondente à classe 0 terá o valor 1 (verdadeiro), indicando que a árvore inferiu a classe 0. No segundo caminho, a comparação $e1 \leq 4$ é falsa e a comparação $e1 \leq 1$ também é falsa, levando à inferência da classe 0. Isso demonstra que, para a mesma classe, podem existir múltiplos caminhos dentro de uma única árvore.

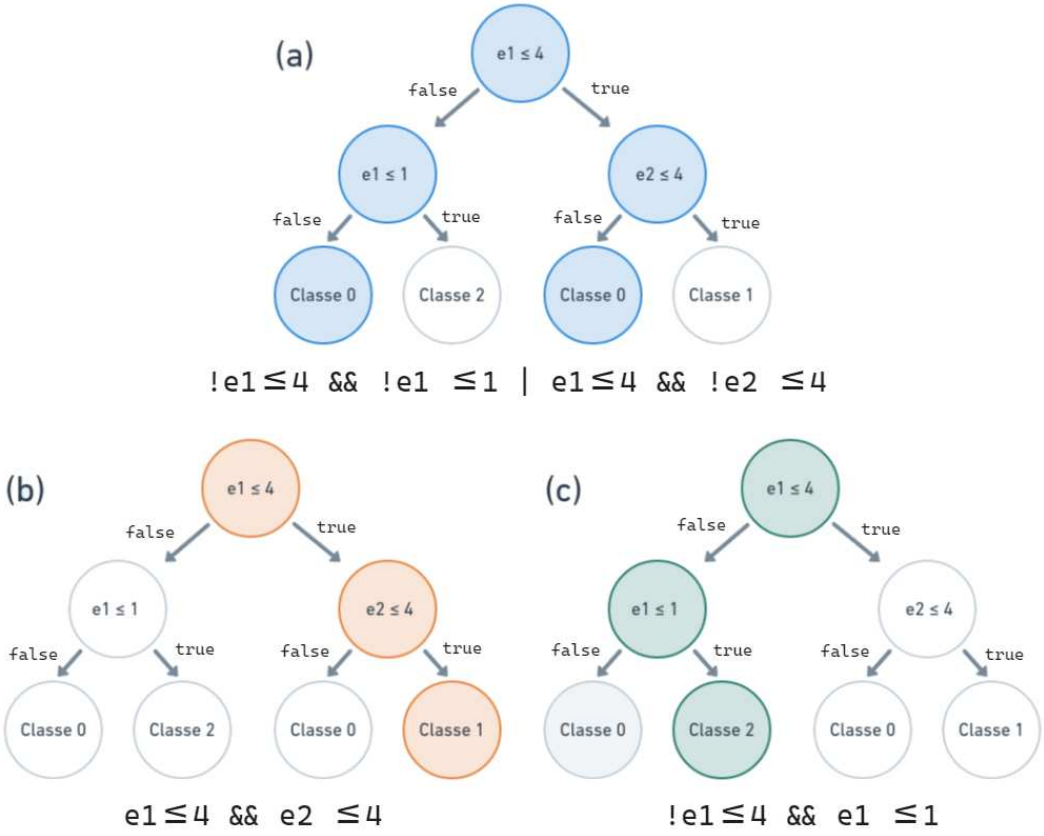
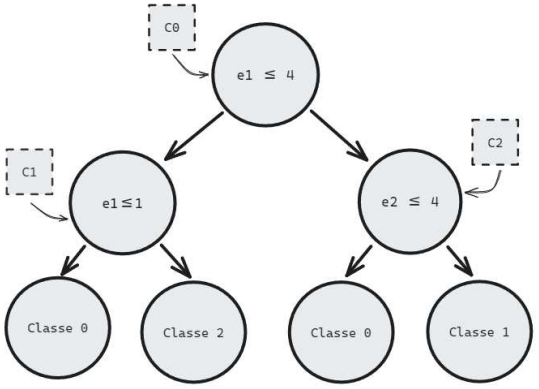


Figura 3.7: Representação de uma árvore de decisão com os caminhos relativos para cada classe em forma de equações booleanas com caminhos visuais e as equações booleanas para inferência das classes.

Na Figura 3.7 (b) e (c), estão representados a mesma árvore, porém indicando o caminho de forma visual e a equação booleana que deve ser satisfeita para que as classes 1 e 2 sejam inferidas.

A Figura 3.8 exemplifica a árvore representada por equações booleanas. Neste exemplo, cada comparador é nomeado como $c0$, $c1$ e $c2$. No código exemplificado, os comparadores $c0$, $c1$ e $c2$ são calculados inicialmente. Após essas comparações, dada uma saída vc de 3 *bits*, criaremos as equações da árvore. Na primeira expressão, o primeiro *bit* da saída (vc) será verdadeiro se o comparador $c1$ for falso ou $c2$ for falso. Na segunda expressão, o segundo *bit* da saída será verdadeiro se o comparador $c0$ for verdadeiro e $c2$ for falso. Finalmente, para que o terceiro *bit* da saída seja verdadeiro, basta que o comparador $c0$ seja falso e o comparador $c1$ seja verdadeiro. É importante destacar que nem todos os resultados das comparações aparecem em todas as equações. Por exemplo, na equação $\sim c1 \mid\sim c2$, a comparação $c0$ não está presente, indicando que o valor de $c0$, verdadeiro ou falso, não importa.




```
assign c0 = e1 ≤ 3'd4;
assign c1 = e1 ≤ 3'd1;
assign c2 = e2 ≤ 3'd4;

assign vc[0] = ~c1 | ~c2;
assign vc[1] = c0 & c2;
assign vc[2] = ~c0 & c1
```

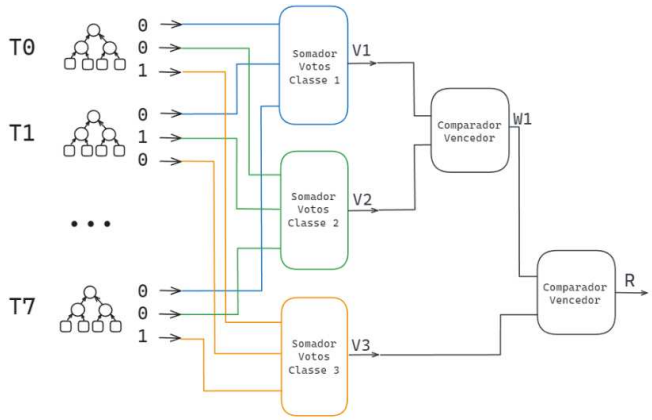
Figura 3.8: Exemplo de uma árvore de decisão representada por equações booleanas.

3.4 Soma de bits para votação

As implementações anteriormente citadas aplicam-se a cada árvore individual dentro do conjunto analisado, apesar de o exemplo ser ilustrativo de apenas uma única árvore. A metodologia descrita é extensível a todas as árvores em uma floresta aleatória. A seguir, delineia-se o procedimento para integrar os resultados em formato *one-hot* de cada árvore, com o objetivo de obter o resultado final da classificação.

Inicialmente, é necessário computar a soma dos votos emitidos por cada árvore. Considerando que a representação dos resultados foi efetuada em formato *one-hot*, implementa-se um circuito somador para cada árvore. Neste contexto, conecta-se o primeiro *bit* de cada saída de cada árvore ao respectivo circuito somador. Devido à natureza do esquema *one-hot*, onde apenas um *bit* é ativo (valor 1) por vez, cada somador acumula o total de votos para cada classe específica.

Após a obtenção dos totais de votos para cada classe, torna-se imperativo comparar esses totais para determinar a classe predominante. Essa comparação é realizada por meio de um circuito comparador, que avalia qual conjunto de votos é maior. Este processo de comparação identifica a classe com a maioria dos votos. Consequentemente, o circuito comparador fornece o resultado final, indicando a classe vencedora na classificação da amostra analisada.




```
//Pseudocódigo para 3 classes e 8 árvores
reg  [2:0] V1;

V1 = T0[0] + T1[0] + T2[0] + ...+ T7[0];
V2 = T0[1] + T1[1] + T2[1] + ...+ T7[1];
V3 = T0[2] + T1[2] + T2[2] + ...+ T7[2];

//Comparador Vencedor
//V1, V2, V3 e W1 → Classe Vencedora
//v1, v2, v3 e w1 → Quantidade de votos
W1    = (v1>v2) ? 1:2;
w1    = (v1>v2) ? v1:v2;
final = (w1>v3) ? W1:3;
```

Figura 3.9: Soma e comparação de votos da *random forest*.

A figura 3.9 demonstra esta fase final do processo de classificação em uma floresta aleatória. Neste esquema, os resultados de cada árvore são representados com três *bits* correspondentes a três classes diferentes. Cada *bit* de saída de todas as árvores é conectado a um dos três somadores de votos, de acordo com a classe que representa: os *bits* representando a classe 1 são conectados ao somador de votos da classe 1, indicado em azul; os da classe 2 ao somador de votos da classe 2, em verde; e os da classe 3 ao somador de votos da classe 3, em amarelo. Estes somadores contabilizam os votos de cada classe com base nas saídas das árvores.

O pseudocódigo ao lado direito da figura 3.9 detalha a lógica subjacente a este processo para um caso de três classes e oito árvores. Três registros (V1, V2 e V3) são usados para armazenar a soma dos votos de cada classe. Para cada classe, os votos correspondentes de todas as árvores (T0 a T7) são somados. Posteriormente, um processo de comparação é realizado para determinar a classe vencedora. Primeiro, compara-se V1 e V2 para determinar o vencedor preliminar (W1) e a quantidade de votos correspondente (w1). Em seguida, esse resultado é comparado com V3 para estabelecer a classe final vencedora. Este mecanismo de somatória e comparação efetua a consolidação dos resultados individuais das árvores em um resultado final unificado.

3.5 BDDs

Um modelo de *random forest* para classificação pode ser representado como um circuito com múltiplas entradas e saídas. Quando mapeamos e sintetizamos uma *random forest* em Verilog, a ferramenta gera um circuito simples que depende apenas dos atributos de entrada e produz poucos *bits* de saída correspondentes à classe inferida. Recentemente, trabalhos como de Murtovi et al. (2022), Nakahara et al. (2017) e a proposta de Silva et al. (2023) têm utilizado diagramas de decisão em *software* para

mapear as *random forests* em funções booleanas. Neste trabalho, adotaremos a abordagem de Silva et al. (2023), baseada em um BDD, para implementar a síntese de circuitos de *random forests* de forma eficiente.

Os Diagramas de Decisão Binárias (*Binary Decision Trees*, BDDs) são uma estrutura de dados utilizada para representar funções booleanas. Em um BDD, cada nó interno representa uma variável de decisão, e cada aresta representa uma possível decisão (verdadeira ou falsa). As folhas da árvore representam os resultados das funções booleanas. O principal objetivo das BDDs é oferecer uma forma compacta e eficiente de representar e manipular funções booleanas, facilitando a execução de operações lógicas complexas e a otimização de circuitos digitais (Bryant, 1986).

Uma das principais vantagens das BDDs é sua capacidade de minimizar a redundância ao representar funções booleanas. Ao contrário de outras representações, como tabelas verdadeiras ou expressões booleanas, as BDDs permitem a redução de subexpressões repetidas, resultando em uma representação mais compacta pois sua representação é canônica, ou seja, só existe uma única representação para cada função. Isso não apenas economiza espaço de armazenamento, mas também melhora a eficiência computacional ao executar operações lógicas. Além disso, os BDDs fornecem métodos eficientes para manipulação e simplificação de funções booleanas, tornando-as ideais para aplicações em verificação formal e síntese de circuitos (Bryant, 1986).

Ao utilizar Diagramas de Decisão Binária (BDDs) para representar árvores de decisão em uma *random forest*, é possível "achatar"o circuito, simplificando sua estrutura. Essa abordagem pode ser particularmente útil para transformar um conjunto de dados simples em uma função booleana que depende apenas das entradas primárias. Nesse contexto, os somadores e os mecanismos de votação são encapsulados juntamente aos nós da árvore de decisão, resultando em funções booleanas mais simplificadas. O BDD tem como entradas os comparadores e encapsula as árvores, a soma e a detecção de maioria em uma única função. Por isso, usamos o termo "achatar"ou deixar planar, traduzindo o termo técnico "*flat*"(em inglês) que é utilizado neste contexto (Silva et al., 2023).

A Figura 3.10(a) ilustra um exemplo que inclui 3 árvores e 4 classes. Portanto, o método proposto utiliza 2 bits para representar cada classe e a classe majoritária requer 2 ou 3 votos. Neste exemplo, a classe 3 emerge como vencedora na árvore 1 e na árvore 2, enquanto a classe 1 é vitoriosa na árvore 3. O resultado final designa a classe 3, com o código 11, como a vencedora geral.

A Figura 3.10(b) mostra o BDD RDSF que representa a floresta aleatória. O BDD possui 2 saídas, nomeadamente F1 e F0, correspondentes ao código binário que representa a classe vencedora. Cada nível do BDD inclui um comparador de entrada. Neste exemplo, o código de saída é 11, ou classe 3, onde as arestas vermelhas representam inversores, as arestas em negrito indicam que a variável de decisão é verdadeira, e as

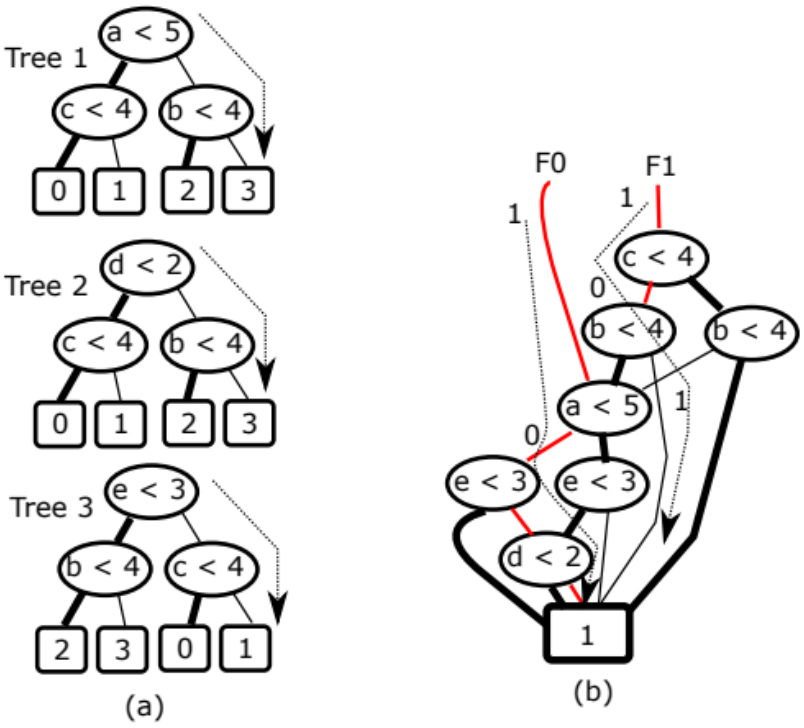


Figura 3.10: Esquema de *hardware* otimizado para a implementação de um *Random Forest* utilizando *Binary Decision Trees* (BDDs). (a) *Random Forest*: 3 Árvores, 4 Classes; (b) 2-*bits* RDSF BDD. Fonte: Silva et al. (2023)

arestas simples denotam que a variável de decisão é falsa. O inversor ocorre apenas em arestas "falsas". Os caminhos do BDD de F1 e F0 até o nó terminal com o código 1 são representados por uma seta pontilhada. O RDSF é percorrido uma vez para executar todas as operações, incluindo a votação, resultando no código 11, ou classe 3.

Para gerar o BDD usamos o algoritmo proposto em (Silva et al., 2023). Primeiro são geradas as equações de cada classe e é construído um BDD para cada classe de cada árvore. Depois usamos a lógica de soma para construir um BDD para cada somatório das classes. Depois um BDD é construído usamos as funções lógica de maioria e finalmente é codificado para compactar o *one-hot code* em um código com $\log n$ *bits* para n classes. Durante este processo, os BDDs intermediários podem ficar muito grandes, o que motiva a aplicar a técnica quando o números de comparadores distintos nas *random forests* não é muito grande (100 ou menos) (Silva et al., 2023).

Um aspecto crucial dos BDDs é a ordenação das variáveis. No BDD, cada variável é um nó comparador da árvore de decisão original e aparece em uma ordem específica que facilita a distribuição do roteamento dos comparadores no circuito. Em uma *random forest*, por outro lado, um comparador pode aparecer em qualquer nível da árvore. O modelo de BDD pode ser implementado usando tabelas, o que ajuda a

preservar a ordem das variáveis. No entanto, em árvores com muitos comparadores diferentes, isso pode resultar em um BDD muito grande ou em BDDs intermediários que demandam muita memória até serem reduzidos ao BDD final. Portanto, técnicas de quantização podem ser usadas para minimizar o número de comparadores necessários.

3.6 Trabalhos Relacionados

Diversos trabalhos implementaram diferentes arquiteturas e abordagens para implementar *Random Forests* e outras árvores de decisão em FPGA. A seguir serão apresentadas algumas abordagens desses trabalhos.

3.6.1 Tabelas de Inferência

Um exemplo de implementação com tabela é apresentado por Zhu et al. (2022), que propôs uma arquitetura de *hardware* para o modelo *Deep Forest*. Este trabalho é pioneiro na aceleração de *Deep Forests* e foi implementado em uma FPGA Intel Stratix V. O design proposto alcança uma aceleração de pelo menos 40x em comparação com um CPU x86 de alto desempenho com 40 núcleos utilizando 32 arvores de profundidade 8. Além disso, apresenta uma eficiência energética comparável aos aceleradores de *hardware* para *Random Forest*, enquanto consome menos recursos de *hardware*. Os conjuntos de dados utilizados foram ADULT e Face Mask Detection. No entanto, detalhes sobre o modelo de memória e a localização dos dados (como DRAM) não foram claramente especificados. Embora o ganho em relação à CPU seja alto, para os *datasets* utilizados não é necessário utilizar este número de árvores. Em *hardware*, as árvores podem ser processadas em paralelo, proporcionando um ganho significativo em comparação com a execução em uma CPU.

Outro trabalho relevante é o de Lin et al. (2019), que introduziu um algoritmo de indução baseado em quantis eficiente e escalável para construção das árvores de Hoeffding, otimizando técnicas de *hardware* específicas para este algoritmo. O sistema de aprendizado de árvores de decisão online foi implementado em FPGA com otimizações de nível de sistema, resultando em uma redução significativa na demanda de memória e computação. O projeto obteve uma melhoria de 0,05% a 12,3% na precisão e uma aceleração de 384x a 1581x no tempo de execução para construção das árvores, ou seja, treinamento, em comparação com o estado da arte. Contudo, a definição do que torna a solução eficiente e escalável, assim como os detalhes de desempenho e metodologia de medição, não foram claramente discutidos. Além disso, apresenta a técnica de *Boosting* utiliza 250 árvores para um *dataset* BUPA com 375 amostras, o que pode indicar um potencial sobreajuste (*overfitting*). É importante notar que o algo-

ritmo *Boosting* é uma técnica distinta das *Random Forests* que são baseadas na técnica de *Bagging*.

3.6.2 Equações Booleanas

Em relação às implementações utilizando equações, Wang et al. (2022) desenvolveram o HardGBM, um *framework* que reduz o tamanho de conjuntos gerados via *Gradient Boosting Machine* (GBM), facilitando a implementação de comitês (*ensembles*) de árvores de regressão. Como já mencionado, o algoritmo *Boosting* não é o mesmo que a *Random Forest*. O HardGBM conseguiu reduzir o tamanho do conjunto inicial mantendo ou até aumentando a precisão da previsão, superando outros métodos de redução em termos de precisão e eficiência.

No entanto, o trabalho não detalha claramente a implementação, deixando dúvidas sobre se o foco foi na inferência ou no treinamento e quanto de recursos foi gasto em cada parte. Além disso, observou-se que um terço dos datasets tiveram melhor acurácia com um número reduzido de árvores, enquanto os demais mostraram pouco ganho mesmo com o aumento significativo do número de árvores. Os *datasets* utilizados no estudo foram *Bank, Telescope, Electricity, Covertype e Person*. Vale mencionar que a implementação do HardGBM exigiu muitos recursos do FPGA, indicando um alto custo de *hardware* para alcançar os resultados desejados.

Outra abordagem utilizando equações foi apresentada por Dávila-Rodríguez et al. (2019), que desenvolveram uma arquitetura FPGA para segmentação de imagens de laranjas em tempo real, utilizando modelos de árvores de decisão. Este trabalho focou na classificação de *pixels* usando nós e saídas binárias, implementando o processo de inferência por meio de equações. Comparadores que descrevem um ramo foram combinados em portas lógicas *AND* com n entradas, onde n representa o número de comparadores necessários. As saídas das portas *AND* foram então agrupadas em uma porta lógica *OR* com m entradas, onde m corresponde ao número de ramos cujo resultado é 1.

3.6.3 Multiplexadores

Por fim, utilizando multiplexadores, Jinguji et al. (2018) propôs uma implementação de *Random Forest* com K-Means *clustering* em FPGA, acelerando o tempo de treinamento e classificação em comparação com implementações em CPU e GPU. O uso do k-Means para quantizar dados antes da classificação e a técnica de compartilhamento de comparadores proposta permitiram a redução do uso de recursos do FPGA sem comprometer a precisão da classificação. No entanto, a comparação de desempenho é dificultada pelo uso de *datasets* pequenos, como *Spambase* e *Traffic*.

Embora os trabalhos citados apresentem contribuições significativas e avanços notáveis, alguns aspectos ainda precisam ser aprimorados. Notavelmente, o uso excessivo de árvores para *datasets* pequenos pode resultar em *overfitting*, o que limita a aplicabilidade prática. Nessa perspectiva, utilizar um número menor de árvores pode ser vantajoso, permitindo a implementação em FPGAs de baixo custo que alcançam resultados similares. Isso não apenas reduz o risco de *overfitting*, mas também torna a solução mais acessível e prática para diversas aplicações. Além disso, não existe comparação entre as diferentes implementações nos trabalhos de FPGA. Em geral, a versão em FPGA é comparada apenas com a versão em CPU ou GPU.

Capítulo 4

Resultados Experimentais

Neste capítulo, apresentamos os principais resultados experimentais deste trabalho. A Seção 4.1 introduz os *datasets* utilizados e suas características. A Seção 4.2 apresenta os resultados das sínteses dos *datasets* nos FPGAs, explorando as ferramentas de síntese e diversas abordagens de projeto. Avaliamos três arquiteturas de FPGA de baixo custo de duas fabricantes diferentes e comparamos suas características e desempenho. A Seção 4.3 apresenta os resultados das sínteses dos *datasets* nos FPGA explorando as ferramentas de síntese e diversas abordagens de projeto.

4.1 Bases de Dados Utilizadas

A escolha dos *datasets* para *benchmarks* em FPGA é um passo crítico para assegurar a relevância e a eficácia dos experimentos realizados. Para esta dissertação, foram selecionados os *datasets*: Susy, Covertype, Adult e Drybean. Cada um desses *datasets* apresenta características específicas que contribuem para uma avaliação abrangente do desempenho das FPGAs em diferentes contextos de classificação. Estes conjuntos de dados estão resumidos na Tabela 4.1, detalhando o número de características, amostras e classes para cada um. Buscamos conjuntos de dados que são utilizados em outros trabalhos com FPGA e que tenham um tamanho relevante para construção das árvores.

	Atributos			
Conjunto de Dados	Numéricos	Categóricos	Amostras	Classes
Covertype	10	44	581K	7
Susy	18	0	5M	2
Adult	6	8	48842	2
Drybean	16	0	13611	7

Tabela 4.1: Bases de Dados: número de atributos, amostras e classes.

4.1.1 Covertype

O *dataset* Covertype, disponível na UCI *Machine Learning Repository*, é amplamente utilizado em pesquisas que envolvem a aceleração de algoritmos de *Random Forest* em GPUs e FPGAs. Este *dataset* foi escolhido devido ao seu alto número de atributos, totalizando 54, o que permite testar a capacidade de manipulação de dados complexos e volumosos pelas FPGAs. Além disso, o Covertype apresenta 7 classes distintas, oferecendo um cenário de classificação multi classes desafiador e realista.

As amostras deste *dataset* foram obtidas a partir de variáveis cartográficas, sem o uso de dados remotamente sensoriados, para prever o tipo de cobertura florestal. Esta característica é particularmente interessante, pois destaca a capacidade do modelo de inferir informações complexas a partir de dados geográficos brutos, uma tarefa comum em aplicações reais de geoprocessamento. A diversidade de classes e a origem dos dados garantem um cenário robusto para testar a eficiência e a flexibilidade das FPGAs na execução de tarefas de classificação (Blackard, 1998).

O *dataset* Covertype tem sido amplamente utilizado em pesquisas para avaliar a performance e o uso de memória em GPUs e dispositivos FPGAs, especialmente no contexto de algoritmos *Random Forest*. No estudo de Van Chu et al. (2021), foi proposto um método de inferência de *decision forest* que transforma a travessia de árvores em travessia de espaços de atributos. Para isso, os autores dividiram cada espaço de atributos em várias regiões com base em limiares.

Neste trabalho, os autores utilizaram o *dataset* Covertype em diversos testes, com um número de árvores variando entre 100 e 200, profundidade entre 5 e 7, quantização de 5 bits e uma acurácia de 69%. Eles demonstraram a eficácia deste *dataset* para tais fins, destacando sua relevância para experimentos de aceleração em *hardware*. Além disso, forneceram uma comparação detalhada entre diferentes aceleradores baseados em FPGA recentemente desenvolvidos.

4.1.2 Susy

O *dataset* Susy, também disponível na UCI *Machine Learning Repository*, foi selecionado principalmente por sua grande quantidade de amostras, totalizando 5 milhões, com 18 atributos e duas classes. Este volume de dados é ideal para testar a escalabilidade e o desempenho das FPGAs na manipulação de grandes conjuntos de dados. A origem dos dados, derivados de simulações de Monte Carlo de propriedades cinemáticas medidas por detectores de partículas em aceleradores, adiciona uma camada de complexidade interessante para a classificação.

Os primeiros 8 atributos do Susy são propriedades cinemáticas diretamente medidas, enquanto os 10 últimos são funções derivadas dessas propriedades, criadas por físicos para ajudar na discriminação entre as duas classes. Este aspecto do *dataset*

permite testar a capacidade das FPGAs em lidar com atributos derivados e potencialmente complexos, que são comuns em muitos cenários de aprendizado de máquina (Whiteson, 2014).

Além disso, a utilização do Susy em *benchmarks* anteriores, como nos estudos de Shah et al. (2022), que utilizaram 25 árvores com profundidade de 15, alcançou uma acurácia de 79,9% e avaliou o desempenho de *Random Forests* em FPGAs em comparação com GPUs. Esses resultados permitem que os FPGAs sejam avaliados de maneira consistente em relação a outros métodos e plataformas de aceleração utilizando esse conjunto de dados.

4.1.3 Adult

O dataset Adult, extraído da base de dados do Censo dos Estados Unidos de 1994, apresenta um desafio único devido à combinação de atributos numéricos e categóricas, totalizando 14, sendo 5 numéricas e 9 categóricas. Com 48.842 amostras, este *dataset* é relativamente pequeno comparado aos demais, o que torna a obtenção de boas acurácias um obstáculo.

O objetivo deste *dataset* é prever se uma pessoa ganha mais de 50 mil dólares por ano, baseado em diversas características demográficas e econômicas. Esta tarefa de classificação binária é representativa de muitos problemas práticos de classificação, onde a interpretação e a manipulação de dados categóricos são cruciais. Testar as FPGAs com este *dataset* ajuda a avaliar a eficácia dos algoritmos implementados em lidar com diferentes tipos de dados e a sua capacidade de generalização a partir de conjuntos de dados limitados.

Além disso, a inclusão do conjunto de dados Adult no *benchmark* permite a comparação com estudos anteriores, como o trabalho de Jansson et al. (2014), onde este conjunto de dados foi utilizado. Nesse estudo, os autores realizaram experimentos de eficiência, coletando métricas em dez pontos de medição variando de 100 a 1000 árvores, com incrementos de 100 árvores para cada conjunto de dados, tanto para os modelos de *Random Forests* quanto para *Extremely Randomized Trees*. A melhor acurácia obtida para o conjunto de dados Adult foi de 85,45%.

4.1.4 Drybean

O *dataset* Drybean, também disponível na UCI *Machine Learning Repository*, contém 16 atributos numéricos e 13.611 amostras, distribuídas em 7 classes. Este *dataset* foi escolhido devido à sua diversidade de classes e à riqueza dos seus atributos, que permitem uma análise detalhada de diferentes tipos de feijões secos. As amostras foram obtidas a partir de imagens de grãos de feijão, que passaram por processos de

segmentação e extração de atributos, resultando em dados que representam tanto a dimensão quanto a forma dos grãos.

Este contexto de classificação é relevante para testar a capacidade das FPGAs em lidar com tarefas de aprendizado de máquina que envolvem a classificação de dados multidimensionais e variados. A diversidade de classes e a especificidade dos atributos extraídos fornecem um cenário robusto para avaliar a precisão e a eficiência dos algoritmos implementados em FPGAs (Koklu and Özkan, 2020).

Além disso, a utilização do conjunto de dados Drybean em *benchmarks* anteriores, como no *framework* HardGBM (Wang et al., 2022), reforça a relevância deste *dataset* para avaliar a eficiência do *hardware* em tarefas de aprendizado de máquina. Nesse estudo, os autores realizaram experimentos com 35, 125 e 250 árvores, alcançando uma acurácia de 92,17%, 92,36% e 92,44%, respectivamente. Embora os autores não tenham especificado a profundidade das árvores, observa-se um pequeno ganho na acurácia com o aumento significativo do número de árvores.

4.2 FPGAs Alvo e Ferramentas de Síntese

Neste estudo, foram escolhidos três dispositivos FPGA de baixo custo, de marcas diferentes, para a realização dos *benchmarks*: Xilinx Artix 7-35T, Intel Cyclone IV EP4CE115 e GoWin GW2AR-18. A seguir, cada FPGA será detalhado em subseções, destacando suas características técnicas, aplicações relevantes e a ferramenta de síntese utilizada.

4.2.1 Xilinx Artix 7-35T

O FPGA Artix-7, integrado na placa de desenvolvimento Artix 7 35T Arty, inclui 20.800 *Look-Up Tables* (LUTs) de 6 entradas, 41.600 *flip-flops*, 90 DSPs, e 1.800 KB de *Block RAM* (BRAM) distribuídos em 50 blocos de 36 Kb. Cada LUT de 6 entradas pode ser configurada para implementar qualquer função lógica de até 6 variáveis. Esses recursos estão organizados em uma arquitetura interna composta por *Configurable Logic Blocks* (CLBs), que por sua vez são divididos em *slices*. Cada CLB contém dois *slices*, e cada *slice* é composto por quatro LUTs e oito *flip-flops*, além de outros recursos como multiplexadores e geradores de sinais de transporte. A placa de desenvolvimento equipada com interfaces de comunicação como USB e *Ethernet*, bem como diversos conectores PMOD, facilita a integração com outros componentes e periféricos, sendo uma escolha versátil para o desenvolvimento de sistemas embarcados e prototipagem rápida e de baixo custo.

Estudos anteriores, como o de Gaikwad et al. (2019), demonstraram a eficácia do Artix 7-35T na implementação de sistemas dedicados de reconhecimento de ativi-

dades humanas (HAR) para dispositivos vestíveis militares inteligentes, utilizando o algoritmo *multilayer* perceptron (MLP). A arquitetura MLP inerente com computação paralela foi implementada de maneira eficiente neste FPGA, mostrando a flexibilidade e o desempenho do dispositivo em tarefas complexas de aprendizado de máquina.

Para a síntese dos projetos no Artix 7-35T, foi utilizado o software Vivado, desenvolvido pela AMD. O Vivado oferece uma suíte completa de ferramentas que inclui entrada de design, síntese, colocação e roteamento, além de verificação/simulação. Essas funcionalidades permitem um fluxo de trabalho otimizado para projetistas que buscam tirar proveito das capacidades adaptativas dos SoCs e FPGAs da AMD.

4.2.2 Intel Cyclone IV EP4CE115

O FPGA Intel Cyclone IV EP4CE115, integrado na placa Altera DE2-115, conta com 114.480 LUTs, 114.480 registradores, 266 DSPs e 66 BRAMs. A série DE2 é reconhecida por suas interfaces abundantes que atendem a diversas necessidades de aplicação, e a placa DE2-115 destaca-se por oferecer um equilíbrio ideal entre baixo custo, baixo consumo de energia e uma rica oferta de capacidades lógicas, de memória e DSP. Este FPGA é especialmente adequado para demandas versáteis e econômicas impulsionadas pela necessidade crescente de acesso a vídeo móvel, voz e dados de alta qualidade.

No contexto de pesquisa, Li et al. (2017) utilizaram o Cyclone IV EP4CE115 para a implementação de blocos neurais baseados na função de ativação sigmoide para aplicações de redes neurais artificiais (ANNs). Este trabalho exemplifica a utilização do FPGA em tarefas de aprendizado de máquina, evidenciando sua capacidade de realizar cálculos complexos de maneira eficiente.

A síntese dos projetos no Cyclone IV EP4CE115 foi realizada utilizando o *software* Quartus da Intel. Este ambiente de design de alto desempenho oferece um fluxo de trabalho intuitivo que abrange desde a entrada e síntese do projeto até a otimização, verificação e simulação, proporcionando aos projetistas uma plataforma ideal para atender às exigências de design da próxima geração.

4.2.3 GoWin GW2AR-18

O FPGA GoWin GW2AR-18, utilizado no kit de desenvolvimento Tang Nano 20K, possui 20.736 *Look-Up Tables* (LUTs), 15.750 registradores, 48 DSPs e 46 *Block RAMs* (BRAMs). Diferente do FPGA Artix-7, que utiliza LUTs de 6 entradas, o GW2AR-18 emprega LUTs de 4 entradas (*LUT4*), o que limita a complexidade das funções lógicas que cada LUT pode implementar em comparação com as LUTs de 6 entradas do Artix-7.

No GW2AR-18, os recursos são organizados de maneira semelhante ao Artix-7, com LUTs e *flip-flops* sendo agrupados em *Configurable Logic Blocks* (CLBs). No entanto, devido ao uso de LUT4, a densidade lógica e a capacidade de implementação de funções complexas por CLB são menores em comparação ao Artix-7. O FPGA GW2AR-18 também é equipado com 2 PLLs (Phase-Locked Loops) para gerenciamento de clock e 48 unidades DSP que suportam multiplicações de 18 bits x 18 bits, o que o torna adequado para aplicações que exigem processamento de sinais digitais.

Comparando com o FPGA Artix-7, o GW2AR-18 do Tang Nano 20K é mais limitado em termos de complexidade lógica e capacidade de armazenamento, devido ao uso de LUT4 em vez de LUT6 e a uma quantidade menor de BRAM. Esses fatores impactam diretamente no desempenho e na densidade de lógica que pode ser implementada no FPGA. Contudo, o Tang Nano 20K se destaca como uma opção acessível para prototipagem e desenvolvimento de sistemas embarcados de menor escala, onde os recursos adicionais do Artix-7 podem não ser necessários.

O kit Tang Nano 20K também inclui um chip BL616 para o download do bitstream e comunicação via UART com o computador, oferecendo uma integração simplificada para o desenvolvimento e a depuração de projetos.

Embora o Tang Nano 20K seja um lançamento recente da Sipeed, estudos prévios como o de Binh et al. (2023) utilizando o Tang Nano 9K demonstram a viabilidade deste FPGA em arquiteturas de IoT seguras e eficientes. A abordagem proposta baseia-se na arquitetura RISC-V personalizada para permitir a criptografia e descriptografia seguras de dados com o algoritmo SHA-256, mostrando a capacidade do FPGA em aplicações de segurança e eficiência energética.

A síntese dos projetos no Sipeed GW2AR-18 foi realizada utilizando o software GOWIN EDA. Este ambiente integrado de design oferece uma solução completa desde a entrada do *design* até a verificação, incluindo a síntese de código, colocação e roteamento, geração de *bitstream* e download para o FPGA. O GOWIN EDA integra a GowinSynthesis para a síntese de design *front-end* e suporta a criação de arquivos RTL e pós-síntese, oferecendo aos engenheiros de design uma plataforma abrangente e fácil de usar.

Comparando-se às ferramentas comerciais da Xilinx e Altera, a ferramenta de síntese Gowin destaca-se por sua simplicidade, proporcionando uma curva de aprendizado rápida e sendo leve e fácil de instalar. Outro aspecto positivo é o tempo de síntese, que é significativamente rápido, permitindo a programação do FPGA em poucos segundos após a conclusão da síntese.

Este estudo, é um dos primeiros trabalhos acadêmicos que avalia as FPGAs da GoWin, realizando uma comparação abrangente entre estas e as FPGAs, bem como as ferramentas de síntese, das duas empresas líderes de mercado com mais de três décadas de experiência, Altera e Xilinx

4.2.4 Comparação de Custo e Recursos FPGAs

Para enriquecer a análise dos FPGAs utilizados, é relevante investigar os aspectos de custo, desempenho e consumo de energia em comparação com outros FPGAs de alto desempenho no mercado. A tabela a seguir apresenta uma comparação detalhada entre os FPGAs de baixo custo utilizados (Xilinx Artix 7-35T, Intel Cyclone IV EP4CE115 e Sipeed GW2AR-18) e alguns dos FPGAs de topo de linha de diferentes marcas, como Xilinx Virtex UltraScale+ e Intel Stratix 10

FPGA	**Custo Aproximado (USD)**	**Recursos (LUTs / BRAM)**	**DSPs**
Sipeed GW2AR-18	$20	20K / 0.8 Mb	48
Xilinx Artix 7-35T	$80	33K / 1.8 Mb	90
Intel Cyclone IV EP4CE115	$150	114K / 3.9 Mb	266
Intel Stratix 10 GX 2800	$2.300	2.8M / 244 Mb	5760
Xilinx Virtex UltraScale+ VU19P	$7.000	4.1M / 224 Mb	3840

Tabela 4.2: Comparação de FPGAs de baixo custo e topo de linha

A análise da Tabela 4.2 revela diferenças significativas entre os FPGAs de baixo custo utilizados nos *benchmarks* e os FPGAs de topo de linha em termos de capacidade de LUTs, memória, DSPs e custo. O Xilinx Artix 7-35T, por exemplo, oferece um bom equilíbrio entre desempenho e eficiência energética, sendo adequado para aplicações embutidas que requerem um consumo reduzido de energia. Em contraste, o Intel Cyclone IV EP4CE115 apresenta uma capacidade de LUTs significativamente maior, mas com um custo mais elevado.

Os FPGAs de topo de linha, como o Xilinx Virtex UltraScale+ e o Intel Stratix 10, oferecem capacidades lógicas e de processamento consideravelmente superiores, refletindo-se também em seu custo. Esses dispositivos são ideais para aplicações de alta performance que requerem grandes volumes de processamento paralelo, como *datacenters* e sistemas avançados de IA.

O Sipeed GW2AR-18, por outro lado, destaca-se por seu custo extremamente baixo, tornando-se uma opção viável para aplicações que necessitam de um FPGA econômico com capacidades lógicas moderadas. Seu consumo de energia é também relativamente baixo, o que o torna adequado para aplicações onde a eficiência energética é crucial.

Para este trabalho, focado na implementação de algoritmos *random forest* em FPGAs, os dispositivos de baixo custo mostraram-se altamente adequados. Uma grande parte dos *datasets* utilizados permitiu alcançar boas acurácias com um número rela-

tivamente pequeno de árvores na floresta, reduzindo significativamente a complexidade e os requisitos de recursos. Isso demonstra que FPGAs como o Xilinx Artix 7-35T, Intel Cyclone IV EP4CE115 e Sipeed GW2AR-18 são capazes de proporcionar desempenho eficiente e suficiente para tarefas de classificação complexas, sem a necessidade dos elevados custos aos FPGAs de topo de linha. Esta eficiência torna os FPGAs de baixo custo uma escolha viável e econômica para a implementação de algoritmos de aprendizado de máquina em uma variedade de aplicações práticas.

4.3 Resultados de síntese

4.3.1 Ferramentas de Síntese, FPGAs Alvo e Conjuntos de Dados

Os Designs Verilog foram avaliados utilizando-se ferramentas de síntese como Vivado, Intel Quartus e Gowin Design 1.9.9 Education, direcionadas às FPGAs Artix-7, Cyclone IV e GW2AR-18, respectivamente.

4.3.2 Validação Experimental em *protoboard* utilizando ESP8266 e Tang Nano 20k

Para verificar a eficácia dos resultados obtidos por meio de simulações computacionais, procedeu-se à montagem de um circuito experimental em uma *protoboard*, utilizando o Tang Nano para realizar a classificação de dados via algoritmo *random forest*. Este método permitiu a visualização prática dos resultados do processo de classificação.

Um diferencial importante ao utilizar um FPGA de baixo custo é a possibilidade de proporcionar fácil acesso aos FPGAs e ilustrar conceitos com exemplos tangíveis. Para fins didáticos, escolhemos conectar o FPGA a um ESP8266 utilizando a IDE Arduino, juntamente com um *dataset* didático e amplamente conhecido, a base de dados Iris. Essa escolha visou ilustrar todo o processo de projeto e execução, facilitando o entendimento dos conceitos envolvidos de maneira prática e acessível.

Inicialmente, programou-se um módulo ESP8266 para armazenar na memória uma série de amostras extraídas dos *datasets* empregados nos testes. A comunicação entre o ESP8266 e o Tang Nano foi estabelecida por meio de uma interface serial. Neste sistema, o ESP8266 transmitia os dados das amostras para o Tang Nano, que já possuía o algoritmo *Random Forest* sintetizado em sua arquitetura.

Após a recepção dos dados, o Tang Nano processava as informações e determinava a classe correspondente de cada amostra. Esta classificação era então enviada de volta ao ESP8266, que exibia os resultados em um *display* OLED. Esta interação

permitiu uma verificação imediata e visual das classificações realizadas pelo sistema embarcado.

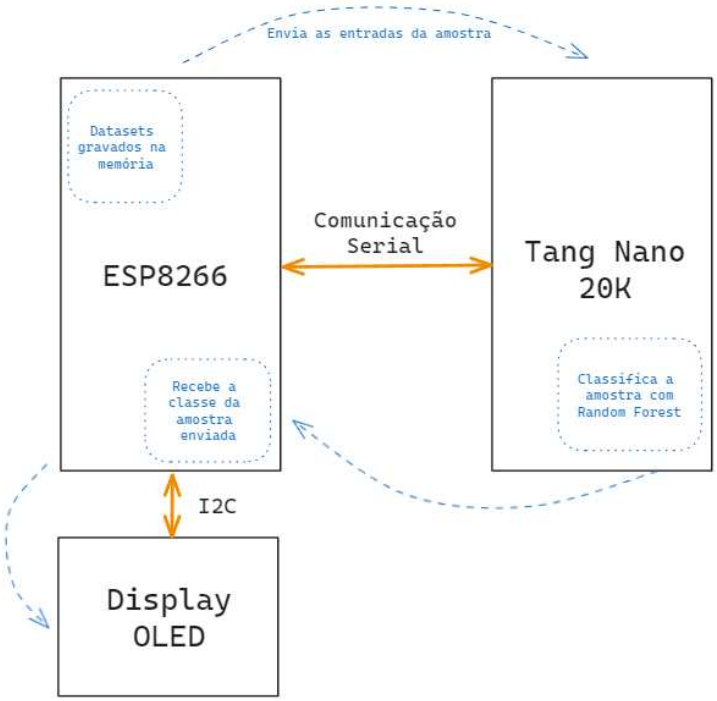


Figura 4.1: Esquema do circuito experimental montado em *protoboard* para validação da classificação por *Random Forest* utilizando Tang Nano e ESP8266, com visualização dos resultados em display OLED.

A Figura 4.1 ilustra o esquema de montagem e operação do circuito implementado. Com base nos resultados esperados, que foram previamente definidos a partir das amostras de teste, foi possível confirmar a precisão do circuito montado. Essa etapa foi crucial para assegurar que a implementação do algoritmo na FPGA, realizada através do processo de síntese no Tang Nano, estava ocorrendo sem omissões ou erros, garantindo a integridade funcional do sistema.

4.3.3 Comparação de quantidade de LUTs utilizadas em cada implementação

Para uma análise do uso de recursos em dispositivos FPGA, foram realizadas sínteses usando quatro diferentes tipos de implementação com os quatro conjuntos de dados previamente introduzidos. As implementações incluem: Multiplexador, Tabela, Condicional e Equação. Cada uma dessas abordagens foi aplicada aos conjuntos de dados em questão utilizando duas ferramentas de síntese: Vivado, da Xilinx, e Gowin, da Sipeed. Nos experimentos as seguir foram utilizadas 7 árvores com profundidade 7,

devido ao tamanho dos *datasets*, todas as árvores geradas pelo *random forest* foram preenchidas em todos os seus níveis, ou seja, são árvores binárias cheias. Tendo isso em vista, sabemos que teremos em cada árvore, 255 nós, sendo que destes 128 são nós folha e os demais nós intermediários.

Adult - 2 classes

Os gráficos da figura 4.2 referentes ao uso de LUTs (*Look-Up Tables*) nos dispositivos FPGA com as diferentes metodologias de implementação revela que a implementação da Tabela apresenta um consumo significativamente maior de LUTs (3224) comparado às demais abordagens. Isso sugere que a implementação baseada em Tabela é menos eficiente em termos de recursos. Em contrapartida, as abordagens Condicional (841), Multiplexador (915), e Equação (949) demonstram um consumo de recursos mais equilibrado e próximo entre si, indicando uma melhor compactação lógica e utilização eficiente dos recursos da FPGA. As três abordagens são bem semelhantes e a provável diferença está na forma que a síntese otimiza o circuito, variando em de 10% no consumo de LUTs entre as três implementações.

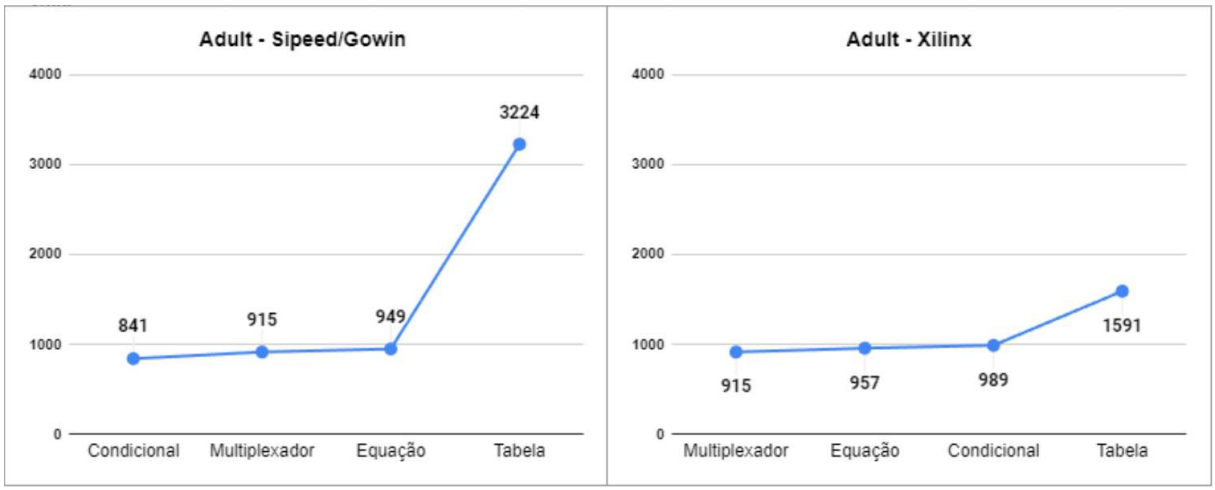


Figura 4.2: Quantidade de LUTs utilizadas em cada implementação utilizando Gowin e Vivado respectivamente para o *dataset* Adult.

Ao comparar com os resultados obtidos utilizando a ferramenta de síntese Xilinx, observa-se um padrão diferente. A metodologia de Tabela, embora ainda elevada em comparação às outras abordagens, apresenta um número consideravelmente menor de LUTs (1591) em relação à síntese realizada com Sipeed. Este resultado pode indicar uma maior eficiência na otimização de recursos pela ferramenta Xilinx para essa implementação específica. Além disso, as implementações Condicional (989), Multiplexador (915) e Equação (957) mantêm uma proximidade nos valores de LUTs utilizadas, tendo uma variação de 8% entre as três abordagens, semelhante ao observado na síntese com Sipeed, reforçando a consistência dessas abordagens na utilização

eficiente dos recursos de FPGA independentemente da ferramenta de síntese.

Covertype - 7 Classes

Ao analisar os gráficos de uso de LUTs na figura 4.3 para o *dataset* Covertype, observa-se novamente uma variação significativa no consumo de recursos entre as diferentes metodologias de implementação e ferramentas de síntese. Utilizando a ferramenta de síntese Sipeed, a implementação baseada em Tabela apresenta o maior consumo de LUTs (5101), seguido pela Equação (2288), Multiplexador (1507), e Condicional (1417). Este padrão de alta utilização de recursos para a metodologia de Tabela é consistente com o observado no *dataset* Adult, indicando que esta abordagem, apesar de possivelmente mais simples em termos de lógica, é menos eficiente em termos de compactação e otimização de recursos. As metodologias Condicional e Multiplexador continuam a demonstrar uma melhor eficiência, destacando-se como opções preferíveis quando se busca minimizar o uso de LUTs.

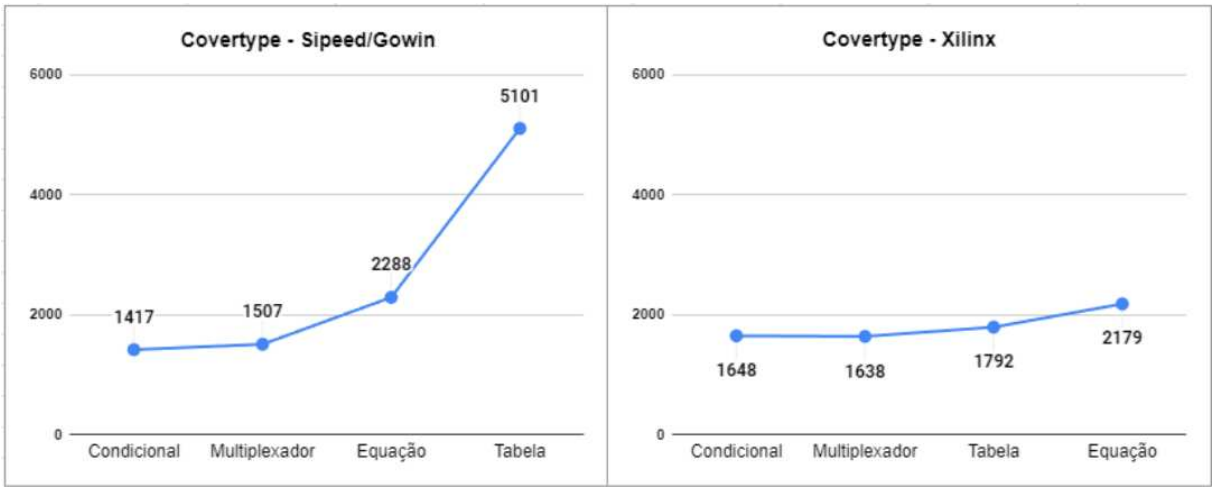


Figura 4.3: Quantidade de LUTs utilizadas em cada implementação utilizando Gowin e Vivado respectivamente para o dataset Covertype.

Quando se considera a ferramenta de síntese Xilinx para o *dataset* Covertype, o padrão de uso de LUTs também apresenta diferenças marcantes. A metodologia de Tabela, embora ainda elevada (1792), é significativamente mais eficiente com Xilinx do que com Sipeed, o Vivado conseguiu capturar bem a tabela e utilizar os registradores de forma mais eficiente. A metodologia Condicional (1648) e Multiplexador (1638) mostram um consumo de recursos muito próximo, indicando uma otimização comparável dessas abordagens pela ferramenta Xilinx. A implementação baseada em Equação (2179), apesar de ser mais eficiente do que a Tabela, ainda demonstra um consumo relativamente alto de LUTs. Este padrão reflete a capacidade da ferramenta Xilinx de otimizar eficientemente implementações complexas, mas também aponta

para a necessidade de considerar a natureza específica do *dataset* ao escolher o tipo de implementação.

DryBean - 7 Classes

Ao analisar os gráficos de uso de LUTs para o *dataset* DryBean na figura 4.4, observa-se novamente uma variação significativa no consumo de recursos entre as diferentes metodologias de implementação e ferramentas de síntese. Utilizando a ferramenta de síntese Sipeed, a metodologia de Tabela continua a apresentar o maior consumo de LUTs (2151), embora menor comparado aos outros *datasets* analisados. As metodologias Condicional (1052), Multiplexador (1125) e Equação (1633) mantêm um uso de recursos relativamente eficiente, com a metodologia Condicional destacando-se como a mais eficiente. Este padrão reflete uma tendência observada nos *datasets* anteriores, onde a implementação por Tabela demonstra uma ineficiência relativa em termos de compactação e otimização de recursos.

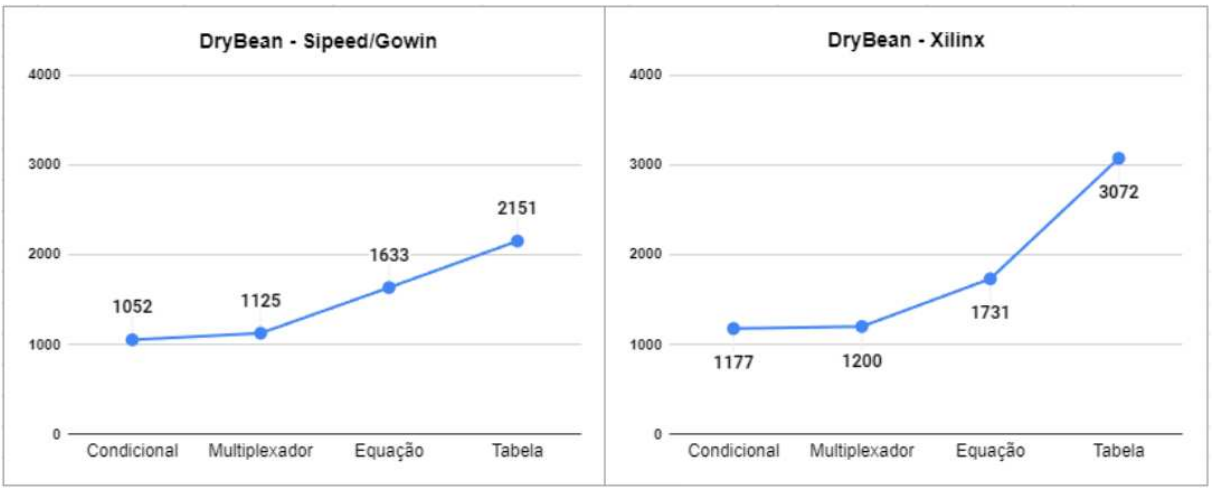


Figura 4.4: Quantidade de LUTs utilizadas em cada implementação utilizando Gowin e Vivado respectivamente para o *dataset* DryBean.

Ao comparar com a ferramenta de síntese Xilinx, observa-se que a implementação de Tabela apresenta um consumo significativamente maior de LUTs (3072) em comparação com Sipeed. No entanto, as outras implementações, Condicional (1177), Multiplexador (1200) e Equação (1731), mantêm-se relativamente próximas em termos de consumo de recursos. Este padrão sugere que, embora a ferramenta Xilinx consiga otimizar eficientemente metodologias mais complexas, a implementação de Tabela continua a ser a menos eficiente. A consistência na eficiência das metodologias Condicional e Multiplexador em ambas as ferramentas reforça a robustez dessas abordagens, destacando-as como escolhas preferenciais para projetos que visam minimizar o uso de recursos de FPGA.

Susy - 2 Classes

A análise dos gráficos de uso de LUTs para o *dataset* SUSY na figura 4.5 revela um padrão interessante, particularmente em relação às metodologias de implementação e ferramentas de síntese. Utilizando a ferramenta de síntese Sipeed, as diferenças no consumo de recursos entre as metodologias são menos discrepantes comparadas aos *datasets* anteriores. As metodologias Condicional (2347), Multiplexador (2381), Equação (2485) e Tabela (2494) apresentam um consumo de LUTs relativamente próximo, sugerindo que a complexidade intrínseca do *dataset* SUSY leva a um uso mais uniforme de recursos, independentemente da abordagem. Isso pode indicar que o *dataset* SUSY, com suas características específicas, exige um nível similar de recursos de implementação, minimizando as diferenças geralmente observadas entre metodologias.

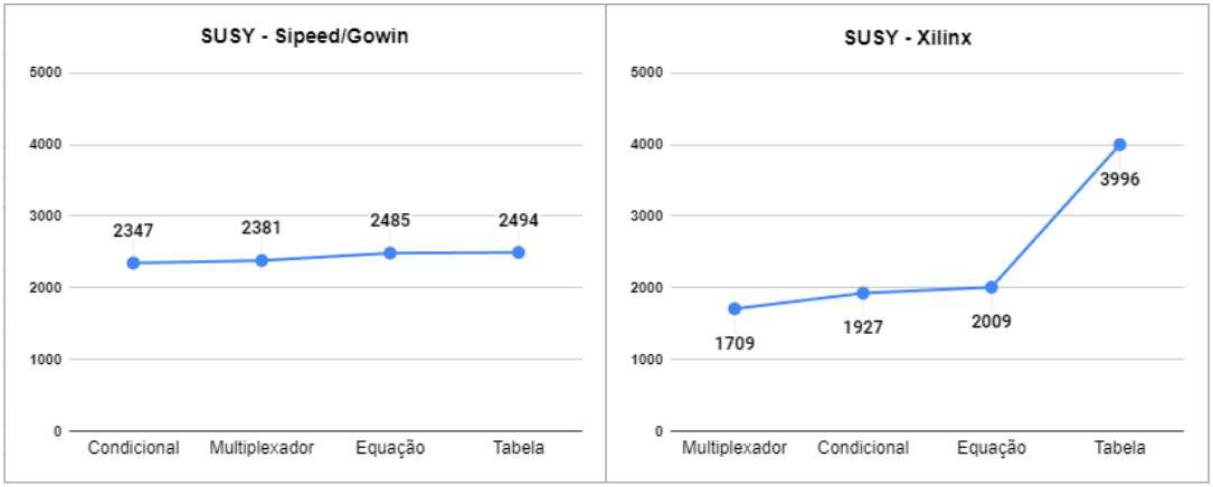


Figura 4.5: Quantidade de LUTs utilizadas em cada implementação utilizando Gowin e Vivado respectivamente para o *dataset* SUSY.

Quando se analisa a síntese com a ferramenta Xilinx, as variações entre as metodologias tornam-se mais evidentes. A implementação baseada em Tabela apresenta o maior consumo de LUTs (3996), significativamente superior às outras metodologias. As implementações Condicional (1927), Multiplexador (1709) e Equação (2009) mostram um consumo de recursos mais equilibrado e consideravelmente menor que a Tabela. Este padrão reflete a eficiência da ferramenta Xilinx em otimizar metodologias mais complexas, mas também destaca a ineficiência relativa da Tabela quando aplicada a *datasets* com alta complexidade como o SUSY. A diferença substancial no consumo de LUTs entre as ferramentas Sipeed e Xilinx para a metodologia de Tabela reforça a importância da escolha da ferramenta de síntese na eficiência final do uso de recursos.

4.3.4 Análise comparativa entre as diferentes implementações e ferramentas de síntese

A tabela 4.3 mostra a quantidade de vezes que cada implementação foi a melhor em cada ferramenta e também mostra o total de LUTs utilizadas em todos os *datasets* entre diferentes implementações em duas ferramentas de síntese, Gowin e Xilinx/Vivado.

	Qtd. vezes melhor	Gowin LUTs	Xilinx/Vivado LUTs	Total por Implementação
Condicional	6	5657	5741	11398
Multiplexador	2	5928	5462	11390
Equação	-	7355	6876	14231
Tabela	-	12970	10451	23421
Total por Ferramenta	-	**31910**	**28530**	-

Tabela 4.3: Comparação de uso de LUTs entre Sipeed/Gowin e Xilinx/Vivado para diferentes implementações

A ferramenta Xilinx/Vivado é, em geral, mais eficiente no uso de LUTs do que a Sipeed/Gowin. O total de LUTs usados pela Xilinx/Vivado (28.530) é menor que o total usado pela Sipeed/Gowin (31.910).

Analisando as diferentes implementações, observa-se que a Tabela utiliza o maior número de LUTs em ambas as ferramentas. No entanto, a ferramenta Xilinx/Vivado consegue capturar esse modo de implementação melhor do que a Sipeed/Gowin, utilizando 10.451 LUTs em comparação com 12.970 LUTs. Condicional (6.657 LUTs) e Multiplexador (5.462 LUTs) por sua vez, mostram um consumo de recursos bastante próximo, indicando que ambas as ferramentas otimizam essas abordagens de forma semelhante, todavia, o Gowin teve uma leve vantagem à descrição com comandos condicionais enquanto o Vivado capturou melhor os multiplexadores explícitos.

No cenário de computação de borda e IoT, faz mais sentido utilizar a implementação de Multiplexador, pois é muito mais eficiente em termos de consumo de recursos. No entanto, se houver muitos dados e a necessidade de alta performance, a implementação por Tabela começa a fazer mais sentido.

Embora tenhamos implementado a Tabela de forma sequencial, essa abordagem pode ser paralelizada, o que traz desafios adicionais, como a sincronização dos registradores, visto que cada árvore pode terminar em tempos diferentes. Vale ressaltar que neste estudo, foi utilizado um tamanho de palavra de 32 bits, mas a normalização para 8 ou 16 bits poderia ter reduzido ainda mais o consumo de LUTs.

Além disso, é importante destacar que a comparação dos atributos durante a indução da *Random Forest* pode ser embarcada ou não. Não embarcar a comparação significa inserir direto no hardware o resultado das comparações dos atributos. Isso

pode oferecer maior flexibilidade e pode ser mais adequada em cenários onde o processamento paralelo e a sincronização dos registradores são menos críticos.

4.3.5 Análise Comparativa da Utilização de LUTs com BDD, Quartis e Kmeans

BDDs são particularmente adequados para lidar com dados categóricos que naturalmente geram decisões binárias. Um dado numérico pode gerar uma ou mais decisões binárias, irá depender do número de ocorrências do atributo nas árvores usando valores distintos de limiares de comparação.

Uma abordagem direta para aplicar BDDs a dados categóricos é utilizar quartis para quantizar ou categorizar os dados numéricos. Esse processo envolve a divisão de um conjunto de dados ordenado em quatro partes iguais, conhecidas como quartis, e a codificação dessas partes utilizando variáveis binárias. Essa técnica simplifica a complexidade dos dados, permitindo que sejam processados de forma mais eficiente por algoritmos de aprendizado de máquina desde de que não haja perda de acurácia.

BDD aplicado ao dataset Adult utilizando quantização por Quartis

Para aplicar BDD ao conjunto de dados Adult utilizando quantização, considerando que o BDD opera de forma mais eficaz com dados categóricos do que numéricos, realizamos os seguintes passos. Primeiramente, quantizamos o conjunto de dados original utilizando quartis. Em seguida, com o dataset quantizado, executamos o algoritmo Random Forest. Posteriormente, aplicamos o BDD e, por fim, geramos o código Verilog a partir do resultado obtido. A figura 4.6 ilustra os passos descritos acima.

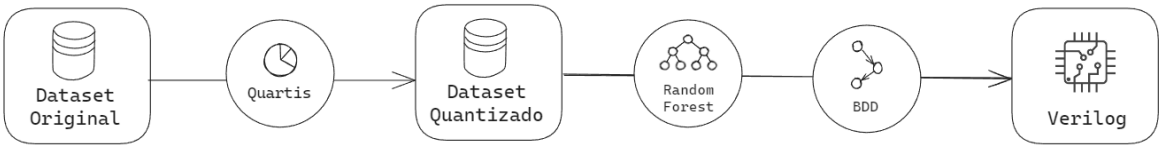


Figura 4.6: Diagrama dos passos para se obter o Verilog a partir de um dataset quantizado aplicando-se o BDD.

Durante a etapa de quantização, inicialmente, cada coluna do conjunto de dados *Adult* foi quantizada e categorizada, abrangendo tanto atributos numéricos quanto categóricos. Cada coluna foi então dividida em quatro classes, necessitando de 2

bits para representar cada classe. Dessa forma, o mapeamento das 14 características resultou em um total de 28 bits.

Após a quantização, analisaram-se diversas configurações de *random forest*, variando de 3 a 7 árvores e com profundidades que variam de 3 até ilimitada. Os algoritmos que utilizam o *dataset* Adult geralmente alcançam uma acurácia média de 82%. Dentre as configurações testadas, a combinação de 7 árvores com profundidade 7 se destacou, alcançando uma precisão de 82,3%. No entanto, essa configuração também resultou em um *overhead* significativo de recursos, exigindo 1317 nós BDD para sua implementação completa.

Considerando que o conjunto de dados foi quantizado no treinamento, é necessário quantizar também as amostras durante a inferência da *random forest*. Para isso, o circuito de inferência é composto por duas partes. A primeira parte é responsável pela quantização, onde se concentra a maior parte dos recursos devido ao uso intensivo de comparadores para categorizar as amostras. A segunda parte é o BDD propriamente dito, que realiza a classificação final, conforme ilustrado na Figura 4.7.

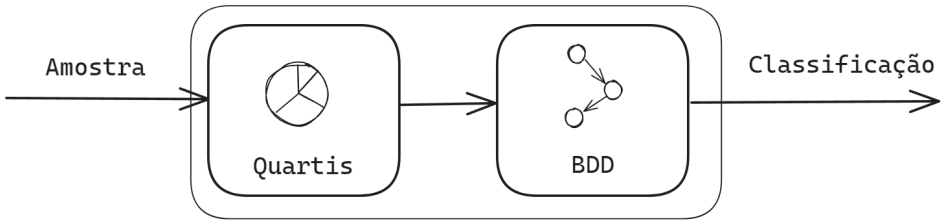


Figura 4.7: Estrutura do circuito de inferência composto pelas etapas de quantização e classificação utilizando BDD.

A partir desta abordagem, podemos verificar no gráfico da figura 4.8, que apesar do BDD ter 1317 nós, que são multiplexadores 2:1, a síntese transformou o BDD em 100 LUTs de 6 entradas. Além disso, ao incluir comparadores dos Quartis, o circuito completo passou a requerer 309 LUTs. Esse número é consideravelmente menor em comparação com o melhor resultado para o RF original de uma etapa com multiplexadores, pois a quantização simplificou o circuito com informações de alto nível que a ferramenta de sintetização não possui.

BDD aplicado ao dataset SUSY utilizando quantização por Quartis

De maneira análoga ao que foi executado para o *dataset* Adult, no conjunto de dados SUSY, também implementamos uma abordagem baseada em quartis. Entretanto, o modificamos a maneira de definir os pontos de corte dos quartis, para isto, selecionamos 6 colunas do conjunto de dados original e aplicamos o algoritmo K-means com $k = 4$ para cada uma dessas colunas testando várias configurações de combinações

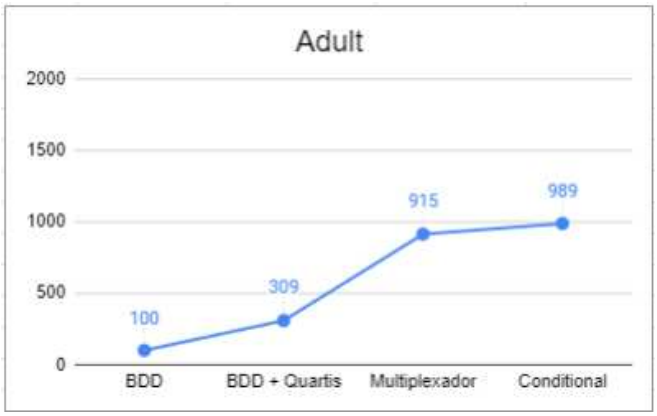


Figura 4.8: Conjunto de dados Adult, onde B é apenas BDD, Q é BDD mais comparadores de quartis, M e C são RF sem BDD com mux ou if/else.

de características até alcançar um resultado interessante. Ao executar a síntese dessa implementação, observou-se o uso de apenas 203 LUTs, o que foi o melhor resultado obtido entre todas as implementações testadas com este conjunto de dados.

BDD aplicado ao dataset SUSY utilizando K-means

Uma abordagem alternativa para a quantização de conjuntos de dados é a utilização do algoritmo K-means para categorizar os dados de maneira semelhante ao método dos quartis. Inicialmente, realizamos este teste, mas, durante a inferência, ao quantizar a amostra, o circuito resultante tornou-se extremamente grande, chegando a utilizar 2985 LUTs (Look-Up Tables). Este crescimento considerável do circuito, ao incluir o K-means, superava os benefícios de sua aplicação, tornando a quantização por quartis uma opção mais vantajosa. A complexidade adicional introduzida pelo K-means não compensava a economia inicial de recursos obtida com o BDD.

Em resposta a este problema, desenvolvemos uma nova estratégia. Ao invés de empregar o K-means e consumir uma grande quantidade de recursos na inferência, optamos por aplicar uma *random forest* aos resultados de cada K-means. Em seguida, geramos (BDDs a partir dos resultados de cada *random forest*. Esta abordagem resultou em um circuito mais eficiente. Assim, obtivemos BDDs intermediários que servem para quantizar a amostra e, posteriormente, um BDD final que realiza a classificação definitiva da amostra.

Utilizando essa abordagem, selecionamos três grupos, cada um composto por duas colunas, totalizando seis colunas selecionadas. Em seguida, aplicamos três algoritmos K-means, cada um com um grupo de duas colunas e um valor de k igual a 5, descartando as colunas restantes.

Para determinar quais colunas seriam utilizadas, realizamos uma busca exploratória. Em 100 iterações, executamos três K-means com 5 clusters, selecionando alea-

toriamente os atributos, e criamos uma *random forest* com o novo conjunto de dados gerado pelos K-means. Avaliamos a acurácia de cada modelo e escolhemos a melhor opção dentre as cinco testadas. A configuração com melhor acurácia foi utilizada para realizar os testes subsequentes.

Essa técnica permitiu uma redução significativa na complexidade e no tamanho do circuito, otimizando a utilização de recursos no FPGA. Embora essa abordagem tenha proporcionado uma economia notável em comparação com a implementação direta do Kmeans, o circuito resultante ainda foi maior do que o obtido com a quantização por quartis.

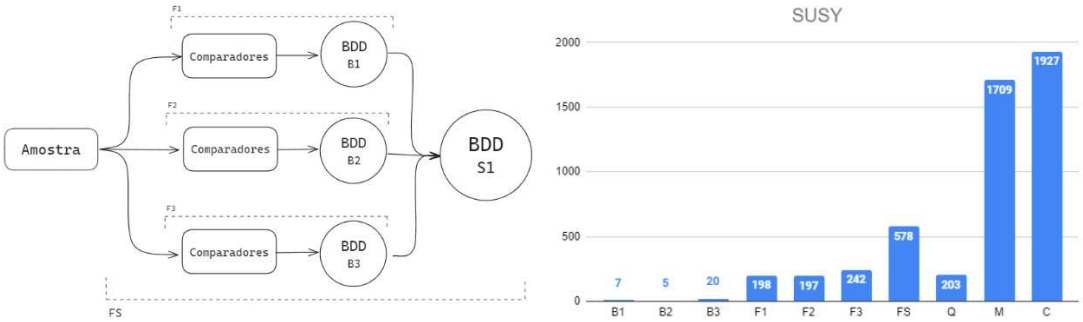


Figura 4.9: Conjunto de dados Susy utilizando BDD, Kmeans, Random Forest de uma e duas etapas.

A Figura 4.9 mostra o esquema de como foram estruturados os BDDs do primeiro e segundo estágio e o gráfico mostra os resultados para o conjunto de dados Susy, onde B1, B2 e B3 representam os BDDs para as três RF da primeira etapa, e S1 é o BDD para a segunda etapa. Os BDDs exigem apenas de 5 a 20 LUTs cada. No entanto, quando se adicionam os comparadores, o tamanho aumenta para 197, 198 e 237 para as três RF da primeira etapa. O circuito completo, que combina as três RFs mais o BDD final, consome 576 LUTs, o que é três vezes menor do que o melhor resultado para a abordagem de uma etapa. No entanto, a abordagem de duas etapas requer 62 comparadores na primeira etapa, resultando em um circuito maior em comparação com a abordagem de quartis (Q) com 206 LUTs (8 vezes menor), que converte as 6 características de entrada em 12 bits usando 3 comparadores por entrada, ou 18 no total.

Apesar da redução de recursos alcançada, o circuito baseado em BDDs com pré-processamento por K-means ainda utilizou menos LUTs do que as abordagens mais simples, como Mux, Equação e Condicional. Essa análise destaca que, embora a combinação de K-means e BDDs ofereça melhorias na eficiência, a quantização por quartis neste cenário continuou sendo a opção mais vantajosa em termos de utilização de recursos para a síntese de FPGAs.

4.3.6 Impacto do Tamanho da Palavra na Utilização de Recursos FPGA

A quantidade de *bits* utilizada para representar os dados a serem utilizados nos modelos de *Random Forest*, impacta diretamente a precisão e a resolução dos dados processados. Maior quantidade de *bits* permite representar números com maior exatidão, o que é essencial para aplicações que exigem alta precisão, como processamento de sinais e algoritmos complexos. No entanto, isso também significa que mais recursos de *hardware* são necessários para armazenar e manipular esses dados, aumentando a utilização de *Look-Up Tables* (LUTs), *flip-flops* e blocos de memória dentro do FPGA.

Além disso, a complexidade dos circuitos lógicos cresce com o aumento do número de *bits*. Operações aritméticas e lógicas tornam-se mais complexas, demandando mais recursos de hardware para sua execução. Caminhos de dados mais longos e complexos dentro do FPGA podem afetar o roteamento e o tempo de propagação dos sinais, resultando em maior latência. Esse aumento na complexidade pode exigir otimizações adicionais durante a síntese para garantir que o design atenda aos requisitos de temporização.

Para analisar o impacto da variação do tamanho da palavra na utilização de LUTs, rodamos o algoritmo *Random Forest* utilizando a implementação condicional com diferentes quantidades de bits (16, 24 e 32 bits) para cada conjunto de dados. Os gráficos a seguir (Figura 4.10) permitem avaliar o quanto de recursos é gasto em cada cenário, oferecendo uma visão clara de como o tamanho da palavra afeta a eficiência da síntese.

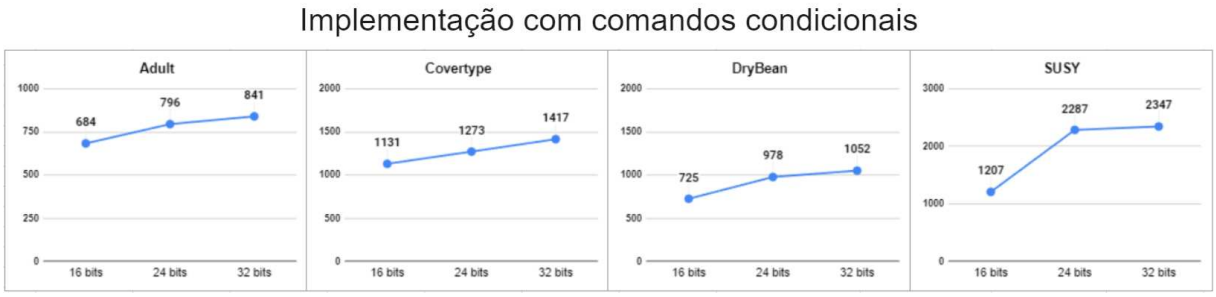


Figura 4.10: Quantidade de LUTs utilizadas para cada conjunto de dados em função do tamanho da palavra (16, 24 e 32 bits).

Esses resultados demonstram que a variação do tamanho da palavra afeta diretamente a utilização de recursos na síntese dos dispositivos FPGA. Palavras maiores oferecem maior acurácia e capacidade de representação de dados, mas exigem mais recursos de *hardware*, como evidenciado pelo aumento no número de LUTs. Portanto, o equilíbrio entre a acurácia necessária e a disponibilidade de recursos de *hardware* é crucial para otimizar a implementação dos algoritmos em FPGAs.

Capítulo 5

Conclusão

Esta pesquisa investigou a implementação de algoritmos de aprendizado de máquina em dispositivos FPGA de baixo custo, especificamente utilizando o algoritmo Random Forest. A análise incluiu diferentes conjuntos de dados (Susy, Covertype, Adult e Drybean) e comparou diversas metodologias de implementação (Multiplexador, Tabela, Condicional e Equação). Os dispositivos FPGA alvo foram Xilinx Artix 7-35T, Intel Cyclone IV EP4CE115 e Sipeed GW2AR-18, com comparações adicionais envolvendo FPGAs de alto desempenho, como Xilinx Virtex UltraScale+ VU19P e Intel Stratix 10 GX 2800.

Os resultados mostraram que os FPGAs de baixo custo podem executar tarefas complexas de classificação com eficiência, utilizando menos recursos de hardware em comparação com os FPGAs de alto desempenho. No entanto, a escolha da metodologia de implementação e da ferramenta de síntese impacta significativamente a utilização de recursos.

Observou-se que a implementação baseada em Tabela tende a ser mais complexa e menos eficiente em termos de uso de LUTs, enquanto as implementações Condicional, Multiplexador e Equação mostraram-se mais eficientes. Em geral, as metodologias Condicional e Multiplexador utilizaram menos LUTs. No entanto, a vantagem da implementação em Tabela é que os dados das árvores ficam todos em memória, não sendo necessário sintetizar novamente caso haja necessidade de mudar a Random Forest.

Além disso, a análise revelou que a quantização por quartis é uma abordagem eficiente para lidar com dados categóricos em FPGAs, utilizando menos recursos do que a quantização por K-means. A análise do impacto do tamanho da palavra na utilização de recursos mostrou que palavras maiores aumentam a demanda por hardware, o que deve ser considerado ao balancear precisão e eficiência.

A validação experimental utilizando protoboard com Tang Nano 20K e ESP8266 demonstrou a viabilidade prática das implementações, confirmando a precisão e a integridade funcional dos algoritmos implementados em FPGA.

5.1 Trabalhos Futuros

Dada a eficiência observada com a quantização por quartis, futuros trabalhos podem focar no aprimoramento das abordagens de quantização, tanto individualmente quanto em conjunto, visando aumentar o desempenho do algoritmo *Random Forest* Bueno et al. (2024b,a). Investigar novas técnicas de quantização pode proporcionar ganhos significativos em termos de uso de recursos e precisão.

Outra área promissora é o desenvolvimento de abordagens que permitam a reprogramação em tempo de execução. A capacidade de ajustar os modelos diretamente no *hardware* durante a operação pode aumentar a flexibilidade e a aplicabilidade dos FPGAs em diversos cenários.

Adicionalmente, é recomendável expandir a pesquisa para FPGAs de alto desempenho. Explorar as capacidades desses dispositivos pode revelar novas oportunidades e limitações, proporcionando uma visão mais abrangente sobre a implementação de algoritmos de aprendizado de máquina em diferentes classes de FPGAs.

A maior parte das técnicas discutidas neste trabalho focam na inferência; entretanto, uma investigação mais profunda sobre o treinamento dos modelos diretamente em FPGA pode abrir novas possibilidades e melhorar a eficiência geral do processo de aprendizado.

Além disso, as técnicas de transformação de árvores de decisão em funções booleanas, utilizadas neste estudo, podem ser avaliadas em outros tipos de modelos, como técnicas de boosting ou redes neurais. Isso pode revelar novas metodologias de implementação que otimizem o uso de recursos.

Por fim, futuros trabalhos devem considerar um equilíbrio entre o uso de memória, DSPs e LUTs. Encontrar o ponto ótimo entre esses recursos é crucial para maximizar a eficiência e o desempenho dos algoritmos implementados em FPGAs.

Referências Bibliográficas

Arapidis, E. (2024). Optimized fpga implementation for random forests for anomaly detection.

Balahur, A. and Turchi, M. (2014). Comparative experiments using supervised learning and machine translation for multilingual sentiment analysis. *Computer Speech & Language*, 28(1):56–75.

Binh, N. T., Kieu-Do, B., Hoang, T.-T., Cong-Kha, P., and Pham-Quoc, C. (2023). Fpga-based secured and efficient lightweight iot edge devices with customized risc-v. In *2023 RIVF International Conference on Computing and Communication Technologies (RIVF)*, pages 31–36.

Blackard, J. (1998). Covertype. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50K5N.

Boutros, A. and Betz, V. (2021). Fpga architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29.

Bragança, L., Canesche, M., Penha, J., Carvalho, W., Comarela, G., Nacif, J. A. M., and Ferreira, R. (2021). An open source custom k-means generator for aws cloud fpga accelerators. In *2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8. IEEE.

Bryant (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

Bueno, W., Silva, O., Nacif, J. A., and Ferreira, R. (2024a). Implementação paralela de múltiplos k-means em gpu. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 37–48. SBC.

Bueno, W., Silva, O., Nacif, J. A., and Ferreira, R. (2024b). Redução de dimensionalidade para árvores aleatórias. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 145–152. SBC.

Canesche, M., Bragança, L., Neto, O. P. V., Nacif, J. A., and Ferreira, R. (2021). Google colab cad4u: Hands-on cloud laboratories for digital design. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Cheng, C. and Bouganis, C.-S. (2013). Accelerating random forest training process using fpga. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–7. IEEE.

Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA c programming*. John Wiley & Sons.

Chong, B. et al. (2021). K-means clustering algorithm: a brief review. *vol*, 4:37–40.

Courtney, M., Breen, M., McMenamin, I., and McNulty, G. (2020). Automatic translation, context, and supervised learning in comparative politics. *Journal of Information Technology & Politics*, 17(3):208–217.

Cruz, E., Adriano, D., Gaudino, E., and Junior, S. (2022). *Sistemas digitais reconfiguráveis: FPGA e VHDL*. Alta Books.

da Silva Alves, M., Silva, L. B., Penha, J., Ferreira, R., and Nacif, J. A. M. (2023). Kcgra–uma arquitetura reconfigurável de domínio específico para k-means. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 25–36. SBC.

Dávila-Rodríguez, I.-A., Nuño-Maganda, M.-A., Hernández-Mier, Y., and Polanco-Martagón, S. (2019). Decision-tree based pixel classification for real-time citrus segmentation on fpga. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE.

de Vos, P., Kirchhoff, M., and Ziener, D. (2020). A complete open source design flow for gowin fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 182–189.

Dhankhad, S., Mohammed, E., and Far, B. (2018). Supervised machine learning algorithms for credit card fraudulent transaction detection: a comparative study. In *2018 IEEE international conference on information reuse and integration (IRI)*, pages 122–125. IEEE.

Dipu Kabir, H. M. and Alam, S. B. (2010). Hardware based realtime, fast and highly secured speech communication using fpga. In *2010 IEEE International Conference on Information Theory and Information Security*, pages 452–457.

Freitas, A. A. (2014). Comprehensible classification models: A position paper. *SIGKDD Explor. Newsl.*, 15(1):1–10.

Gaikwad, N. B., Tiwari, V., Keskar, A., and Shivaprakash, N. C. (2019). Efficient fpga implementation of multilayer perceptron for real-time human activity classification. *IEEE Access*, 7:26696–26706.

Gunasekaran, K., Baskar, R., Dhanagopal, R., and Elangovan, K. (2021). Classification of fpga based network traffic using machine learning. In *Journal of Physics: Conference Series*, volume 1964, page 062008. IOP Publishing.

Hartshorn, S. (2016). Machine learning with random forests and decision trees: A visual guide for beginners. *Kindle edition*.

Jansson, K., Sundell, H., and Boström, H. (2014). gpurf and gpuert: efficient and scalable gpu algorithms for decision tree ensembles. In *2014 IEEE international parallel & distributed processing symposium workshops*, pages 1612–1621. IEEE.

Jinguji, A., Sato, S., and Nakahara, H. (2018). An fpga realization of a random forest with k-means clustering using a high-level synthesis design. *IEICE TRANSACTIONS on Information and Systems*, 101(2):354–362.

Kaggle (2021). State of data science and machine learning. `https://www.kaggle.com/kaggle-survey-2021`.

Koklu, M. and Özkan, I. A. (2020). Dry Bean. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50S4B.

Laurent, H. and Rivest, R. L. (1976). Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17.

Leo Breiman, Jerome Friedman, C. J. S. R. O. (1984). *Classification and Regression Trees*. Chapman and Hall/CRC.

Li, Z., Huang, Y.-J., and Lin, W.-C. (2017). Fpga implementation of neuron block for artificial neural network. In *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2.

Lin, Z., Sinha, S., and Zhang, W. (2019). Towards efficient and scalable acceleration of online decision tree learning on fpga. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 172–180. IEEE.

Liu, S., Lau, F. C., and Schafer, B. C. (2019). Accelerating fpga prototyping through predictive model-based hls design space exploration. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6.

Loh, W.-Y. (2011). Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(1):14–23.

Marulanda Lopez, J. F., de Brito Neto, W. B., and dos Santos Ferreira, R. (2024). Machine learning approach to support taxonomic discrimination of mayflies species based on morphologic data. *Neotropical Entomology*, pages 1–8.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Murtovi, A., Bainczyk, A., and Steffen, B. (2022). Forest gump: a tool for explanation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 314–331. Springer.

Nakahara, H., Jinguji, A., Sato, S., and Sasao, T. (2017). A random forest using a multi-valued decision diagram on an fpga. In *2017 IEEE 47th international symposium on multiple-valued logic (ISMVL)*, pages 266–271. IEEE.

Ordoñez, E. D. M. (2003). *Projeto, desempenho e aplicações de sistemas digitais em circuitos programáveis (FPGAs)*. Cesar Giacomini Penteado.

Ordoñez, E. D. M., Penteado, C. G., and da Silva, A. C. R. (2005). *Microcontroladores e FPGAs: aplicações em automação*. Novatec Editora.

Owaida, M., Zhang, H., Zhang, C., and Alonso, G. (2017). Scalable inference of decision tree ensembles: Flexible design for cpu-fpga platforms. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE.

Penha, J., da Silva, A. K., Barros, O., Moreira, I., Nacif, J. A. M., and Ferreira, R. (2023). Avaliação de estilos de código para árvores de decisão em gpu com microbenchmarks. In *Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 277–288. SBC.

Penha, J. C., Bragança, L., Coelho, K., Canesche, M., Silva, J., Comarela, G., Nacif, J. A. M., and Ferreira, R. (2018). A gpu/fpga-based k-means clustering using a parameterized code generator. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 61–69. IEEE.

Pham-Quoc, C., Pham-Dinh, T., and Kieu-Do-Nguyen, B. (2024). Efficient random forest acceleration for edge computing platforms with fpga technology. *Journal of Advances in Information Technology*, 15(2).

Prince, S. J. (2023). *Understanding Deep Learning*. MIT press.

Ruiz, M., Sidler, D., Sutter, G., Alonso, G., and López-Buedo, S. (2019). Limago: An fpga-based open-source 100 gbe tcp/ip stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 286–292. IEEE.

Rushton, A. (2011). *VHDL for logic synthesis*. John Wiley & Sons.

Saqib, F., Dutta, A., Plusquellic, J., Ortiz, P., and Pattichis, M. S. (2013). Pipelined decision tree classification accelerator implementation in fpga (dt-caif). *IEEE Transactions on Computers*, 64(1):280–285.

Senagi, K. and Jouandeau, N. (2022). Parallel construction of random forest on gpu. *The Journal of Supercomputing*, 78(8):10480–10500.

Shah, M., Neff, R., Wu, H., Minutoli, M., Tumeo, A., and Becchi, M. (2022). Accelerating random forest classification on gpu and fpga. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11.

Silva, A., Silva, O., Moreira, I., Nacif, J. A., and Ferreira, R. (2024). Implementações eficientes de random forest em fpga de baixo custo para internet das coisas e computação de borda. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 49–60. SBC.

Silva, L., Penha, J., Ribeiro, D., Silva, A., Nacif, J., and Ferreira, R. (2022). Hpyc-fpga - integração de aceleradores em fpga de alto desempenho com python para jupyter notebooks. In *Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 193–204, Porto Alegre, RS, Brasil. SBC.

Silva, O. A. B., Silva, A. K. C., Moreira, G. S., Nacif, J. A. M., and Ferreira, R. S. (2023). Rdsf: Everything at same place all at once - a random decision single forest. In *2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–6.

Smith, D. J. (1996). Vhdl & verilog compared & contrasted—plus modeled example written in vhdl, verilog and c. In *Proceedings of the 33rd annual Design Automation Conference*, pages 771–776.

Struharik, J. (2011). Implementing decision trees in hardware. In *2011 IEEE 9th International Symposium on Intelligent Systems and Informatics*, pages 41–46.

Teixeira, J. (2019). *O que é inteligência artificial*. E-Galáxia.

Van Chu, T., Kitajima, R., Kawamura, K., Yu, J., and Motomura, M. (2021). A high-performance and flexible fpga inference accelerator for decision forests based on prior feature space partitioning. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10.

Wang, H., Wu, Z., Wang, X., Bian, L., and Jin, H. (2022). Hardgbm: A framework for accurate and hardware-efficient gradient boosting machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Whiteson, D. (2014). SUSY. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C54606.

Wolf, C., Glaser, J., and Kepler, J. (2013). Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97.

Zaki, M. J. and Meira, W. (2014). *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press.

Zhu, M., Luo, J., Mao, W., and Wang, Z. (2022). An efficient fpga-based accelerator for deep forest. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3334–3338. IEEE.